Основы программирования на C#

1. Лекция: Visual Studio .Net, Framework .Net: версия для печати и PDA
Бренд .Net. Visual Studio .Net - открытая среда разработки. Каркас Framework .Net. Библиотека классов FCL - статический компонент каркаса. Общеязыковая исполнительная среда CLR - динамический компонент каркаса. Управляемый код. Общеязыковые спецификации CLS и совместимые модули.

Имя .Net

Имена нынешнего поколения продуктов от Microsoft сопровождаются окончанием .Net (читается Dot Net), отражающим видение Microsoft современного коммуникативного мира. Компьютерные сети объединяют людей и технику. Человек, работающий с компьютером или использующий мобильный телефон, естественным образом становится частью локальной или глобальной сети. В этой сети используются различные специальные устройства, начиная от космических станций и кончая датчиками, расположенными, например, в гостиницах и посылающими информацию об объекте всем мобильным устройствам в их окрестности. В глобальном информационном мире коммуникативная составляющая любых программных продуктов начинает играть определяющую роль.

В программных продуктах .Net за этим именем стоит вполне конкретное содержание, которое предполагает, в частности, наличие открытых стандартов коммуникации, переход от создания монолитных приложений к созданию компонентов, допускающих повторное использование в разных средах и приложениях. Возможность повторного использования уже созданных компонентов и легкость расширения их функциональности - все это непременные атрибуты новых технологий. Важную роль в этих технологиях играет язык XML, ставший стандартом обмена сообщениями в сети.

Не пытаясь охватить все многообразие сетевого взаимодействия, рассмотрим реализацию новых идей на примере Visual Studio .Net - продукта, важного для разработчиков.

Visual Studio .Net - открытая среда разработки

Среда разработки Visual Studio .Net - это уже проверенный временем программный продукт, являющийся седьмой версией Студии. Но новинки этой версии, связанные с идеей .Net, позволяют считать ее принципиально новой разработкой, определяющей новый этап в создании программных продуктов. Выделю две важнейшие, на мой взгляд, идеи:

- открытость **для языков программирования;**
- принципиально новый подход к построению **каркаса среды - Framework .Net**.

Открытость

Среда разработки теперь является открытой языковой средой. Это означает, что наряду с языками программирования, включенными в среду фирмой Microsoft - **Visual C++ .Net** (с управляемыми расширениями), **Visual C# .Net**, **J# .Net**, **Visual Basic .Net**, - в среду могут добавляться любые языки программирования, компиляторы которых создаются другими фирмами-производителями. Таких расширений среды **Visual Studio** сделано уже достаточно много, практически они существуют для всех известных языков - **Fortran** и **Cobol**, **RPG** и **Component Pascal**, **Oberon** и **SmallTalk**. Я у себя на компьютере включил в среду компилятор одного из лучших объектных языков - языка **Eiffel**.

Открытость среды не означает полной свободы. Все разработчики компиляторов при

включении нового языка в среду разработки должны следовать определенным ограничениям. Главное ограничение, которое можно считать и главным достоинством, состоит в том, что все языки, включаемые в среду разработки Visual Studio .Net, должны использовать единый каркас - Framework .Net. Благодаря этому достигаются многие желательные свойства: легкость использования компонентов, разработанных на различных языках; возможность разработки нескольких частей одного приложения на разных языках; возможность бесшовной отладки такого приложения; возможность написать класс на одном языке, а его потомков - на других языках. Единый каркас приводит к сближению языков программирования, позволяя вместе с тем сохранять их индивидуальность и имеющиеся у них достоинства. Преодоление языкового барьера - одна из важнейших задач современного мира. Благодаря единому каркасу, Visual Studio .Net в определенной мере решает эту задачу в мире программистов.

Framework .Net - единый каркас среды разработки

В каркасе Framework .Net можно выделить два основных компонента:

- статический - FCL (Framework Class Library) - библиотеку классов каркаса;
- динамический - CLR (Common Language Runtime) - общеязыковую исполнительную среду.

Библиотека классов FCL - статический компонент каркаса

Понятие каркаса приложений - Framework Applications - появилось достаточно давно; по крайней мере оно широко использовалось еще в четвертой версии Visual Studio. Десять лет назад, когда я с Ильмиром писал книгу [В.А. Биллиг, И.Х. Мусикаев "Visual C++, 4-я версия. Книга для программистов"], для нас это было еще новое понятие. Мы подробно обсуждали роль библиотеки классов MFC (Microsoft Foundation Classes) как каркаса приложений Visual C. Несмотря на то, что каркас был представлен только статическим компонентом, уже тогда была очевидна его роль в построении приложений. Уже в то время важнейшее значение в библиотеке классов MFC имели классы, задающие архитектуру строящихся приложений. Когда разработчик выбирал один из возможных типов приложения, например, архитектуру Document-View, то в его приложение автоматически встраивались класс Document, задающий структуру документа, и класс View, задающий его визуальное представление. Класс Form и классы, задающие элементы управления, обеспечивали единый интерфейс приложений. Выбирая тип приложения, разработчик изначально получал нужную ему функциональность, поддерживаемую классами каркаса. Библиотека классов поддерживала и более традиционные для программистов классы, задающие расширенную систему типов данных, в частности, динамические типы данных - списки, деревья, коллекции, шаблоны.

За прошедшие 10 лет роль каркаса в построении приложений существенно возросла - прежде всего, за счет появления его динамического компонента, о котором чуть позже поговорим подробнее. Что же касается статического компонента - библиотеки классов, то и здесь за десять лет появился ряд важных нововведений.

Единство каркаса

Каркас стал единым для всех языков среды. Поэтому, на каком бы языке программирования ни велась разработка, она использует классы одной и той же библиотеки. Многие классы библиотеки, составляющие общее ядро, используются всеми языками. Отсюда единство интерфейса приложения, на каком бы языке оно не разрабатывалось, единство работы с коллекциями и другими контейнерами данных, единство связывания с различными

хранилищами данных и прочая универсальность.

Встроенные примитивные типы

Важной частью библиотеки FCL стали классы, задающие примитивные типы - те типы, которые считаются встроенными в язык программирования. Типы каркаса покрывают все множество встроенных типов, встречающихся в языках программирования. Типы языка программирования проецируются на соответствующие типы каркаса. Тип, называемый в языке Visual Basic - Integer, а в языке C# - int, проецируется на один и тот же тип каркаса System.Int32. В каждом языке программирования, наряду с "родными" для языка названиями типов, разрешается пользоваться именами типов, принятыми в каркасе. Поэтому, по сути, все языки среды разработки могут пользоваться единой системой встроенных типов, что, конечно, способствует облегчению взаимодействия компонентов, написанных на разных языках.

Структурные типы

Частью библиотеки стали не только простые встроенные типы, но и структурные типы, задающие организацию данных - строки, массивы, перечисления, структуры (записи). Это также способствует унификации и реальному сближению языков программирования.

Архитектура приложений

Существенно расширился набор возможных архитектурных типов построения приложений. Помимо традиционных Windows- и консольных приложений, появилась возможность построения Web-приложений. Большое внимание уделяется возможности создания повторно используемых компонентов - разрешается строить библиотеки классов, библиотеки элементов управления и библиотеки Web-элементов управления. Популярным архитектурным типом являются Web-службы, ставшие сегодня благодаря открытому стандарту одним из основных видов повторно используемых компонентов. Для языков C#, J#, Visual Basic, поддерживаемых Microsoft, предлагается одинаковый набор из 12 архитектурных типов приложений. Несколько особняком стоит Visual C++, сохраняющий возможность работы не только с библиотекой FCL, но и с библиотеками MFC и ATL, и с построением соответствующих MFC и ATL-проектов. Компиляторы языков, поставляемых другими фирмами, создают проекты, которые удовлетворяют общим требованиям среды, сохраняя свою индивидуальность. Так, например, компилятор Eiffel допускает создание проектов, использующих как библиотеку FCL, так и собственную библиотеку классов.

Модульность

Число классов библиотеки FCL велико (несколько тысяч). Поэтому понадобился способ их структуризации. Логически классы с близкой функциональностью объединяются в группы, называемые пространством имен (Namespace). Для динамического компонента CLR физической единицей, объединяющей классы и другие ресурсы, является сборка (assembly).

Основным пространством имен библиотеки FCL является пространство System, содержащее как классы, так и другие вложенные пространства имен. Так, уже упоминавшийся примитивный тип Int32 непосредственно вложен в пространство имен System и его полное имя, включающее имя пространства - System.Int32.

В пространство System вложен целый ряд других пространств имен. Например, в пространстве System.Collections находятся классы и интерфейсы, поддерживающие работу с коллекциями объектов - списками, очередями, словарями. В пространство System.Collections, в свою очередь, вложено пространство имен Specialized, содержащие классы со специализацией, например, коллекции, элементами которых являются только

строки. Пространство System.Windows.Forms содержит классы, используемые при создании Windows-приложений. Класс Form из этого пространства задает форму - окно, заполняемое элементами управления, графикой, обеспечивающее интерактивное взаимодействие с пользователем.

По ходу курса мы будем знакомиться со многими классами, принадлежащими различным пространствам имен библиотеки FCL.

Общеязыковая исполнительная среда CLR - динамический компонент каркаса

Наиболее революционным изобретением Framework .Net явилось создание **исполнительной среды** CLR. С ее появлением процесс написания и выполнения приложений становится принципиально другим. Но обо всем по порядку.

Двухэтапная компиляция. Управляемый модуль и управляемый код

Компиляторы языков программирования, включенные в Visual Studio .Net, создают модули на промежуточном языке MSIL (Microsoft Intermediate Language), называемом далее просто - IL. Фактически компиляторы создают так называемый управляемый модуль - переносимый исполняемый файл (Portable Executable или PE-файл). Этот файл содержит код на IL и метаданные - всю необходимую информацию как для CLR, так и конечных пользователей, работающих с приложением. О метаданных - важной новинке Framework .Net - мы еще будем говорить неоднократно. В зависимости от выбранного типа проекта, PE-файл может иметь расширения exe, dll, mod или mdl.

Заметьте, PE-файл, имеющий расширение exe, хотя и является exe-файлом, но это не совсем обычный исполняемый Windows файл. При его запуске он распознается как специальный PE-файл и передается CLR для обработки. Исполнительная среда начинает работать с кодом, в котором специфика исходного языка программирования исчезла. Код на IL начинает выполняться под управлением CLR (по этой причине **код** называется управляемым). Исполнительную среду можно рассматривать как своеобразную виртуальную IL-машину. Эта машина транслирует "на лету" требуемые для исполнения участки кода в команды реального процессора, который в действительности и выполняет код.

Виртуальная машина

Отделение каркаса от студии явилось естественным шагом. Каркас Framework .Net перестал быть частью студии, а стал надстройкой над операционной системой. Теперь компиляция и создание PE-модулей на IL отделены от выполнения, и эти процессы могут быть реализованы на разных платформах. В состав CLR входят трансляторы JIT (Just In Time Compiler), которые и выполняют трансляцию IL в командный код той машины, где установлена и функционирует исполнительная среда CLR. Конечно, в первую очередь Microsoft реализовала CLR и FCL для различных версий Windows, включая Windows 98/Me/NT 4/2000, 32 и 64-разрядные версии Windows XP и семейство .Net Server. Для операционных систем Windows CE и Palm разработана облегченная версия Framework .Net.

В 2001 году ЕСМА (Европейская ассоциация производителей компьютеров) приняла язык программирования C#, CLR и FCL в качестве стандарта, так что Framework .Net уже функционирует на многих платформах, отличных от Windows. Он становится свободно распространяемой виртуальной машиной. Это существенно расширяет сферу его применения. Производители различных компиляторов и сред разработки программных продуктов предпочитают теперь также транслировать свой код в IL, создавая модули в соответствии со спецификациями CLR. Это обеспечивает возможность выполнения их кода

на разных платформах.

Microsoft использовала получивший широкое признание опыт виртуальной машины Java, улучшив процесс за счет того, что, в отличие от Java, промежуточный код не интерпретируется исполнительной средой, а компилируется с учетом всех особенностей текущей платформы. Благодаря этому создаются высокопроизводительные приложения.

Следует отметить, что CLR, работая с IL-кодом, выполняет достаточно эффективную оптимизацию и, что не менее важно, защиту кода. Зачастую нецелесообразно выполнять оптимизацию на уровне создания IL-кода - она иногда может не улучшить, а ухудшить ситуацию, не давая CLR провести оптимизацию на нижнем уровне, где можно учесть даже особенности процессора.

Дизассемблер и ассемблер

Если у вас есть готовый PE-файл, то иногда полезно анализировать его IL-код и связанные с ним метаданные. В состав Framework SDK входит дизассемблер - ildasm, выполняющий дизассемблирование PE-файла и показывающий метаданные, а также IL-код с комментариями в наглядной форме. Мы иногда будем пользоваться результатами дизассемблирования. У меня на компьютере кнопка, вызывающая дизассемблер, находится на панели, где собраны наиболее часто используемые мной приложения. Вот путь к папке, в которой обычно находится дизассемблер:

```
C:\Program Files\Microsoft Visual Studio .Net\ FrameworkSDK\Bin\ildasm.exe
```

Профессионалы, предпочитающие работать на низком уровне, могут программировать на языке ассемблера IL. В этом случае в их распоряжении будет вся мощь библиотеки FCL и все возможности CLR. У меня на компьютере путь к папке, где находится ассемблер, следующий:

```
C:\WINDOWS\Microsoft.Net\Framework\v1.1.4322\ilasm.exe
```

В этом курсе к ассемблеру мы обращаться не будем - я упоминаю о нем для полноты картины.

Метаданные

Переносимый исполняемый PE-файл является самодокументируемым файлом и, как уже говорилось, содержит и код, и метаданные, описывающие код. Файл начинается с манифеста и включает в себя описание всех классов, хранимых в PE-файле, их свойств, методов, всех аргументов этих методов - всю необходимую CLR информацию. Поэтому помимо PE-файла не требуется никаких дополнительных файлов и записей в реестр - вся нужная информация извлекается из самого файла. Среди классов библиотеки FCL имеется класс Reflection, методы которого позволяют извлекать необходимую информацию. Введение метаданных - не только важная техническая часть CLR, но это также часть новой идеологии разработки программных продуктов. Мы увидим, что и на уровне языка C# самодокументированию уделяется большое внимание.

Мы увидим также, что при проектировании класса программист может создавать собственные атрибуты, добавляемые к метаданным PE-файла. Клиенты этого класса могут, используя класс Reflection, получать эту дополнительную информацию, и на ее основании принимать соответствующие решения.

На рис. 1.1 показаны результаты дизассемблирования PE-файла простого консольного приложения с именем Account, включающего три класса: Account, Testing и Class1.

Дизассемблер структурирует информацию, хранимую в метаданных, и показывает ее в типичном формате дерева. Как обычно, это дерево можно сжимать или раскрывать, демонстрируя детали класса. Значки, приписываемые каждому узлу дерева, характеризуют тип узла - класс, свойство, метод, описание. Двойной щелчок кнопки мыши на этом узле позволяет раскрыть его. При раскрытии метода можно получить его код. На рис. 1.1 показан код метода add из класса Account.

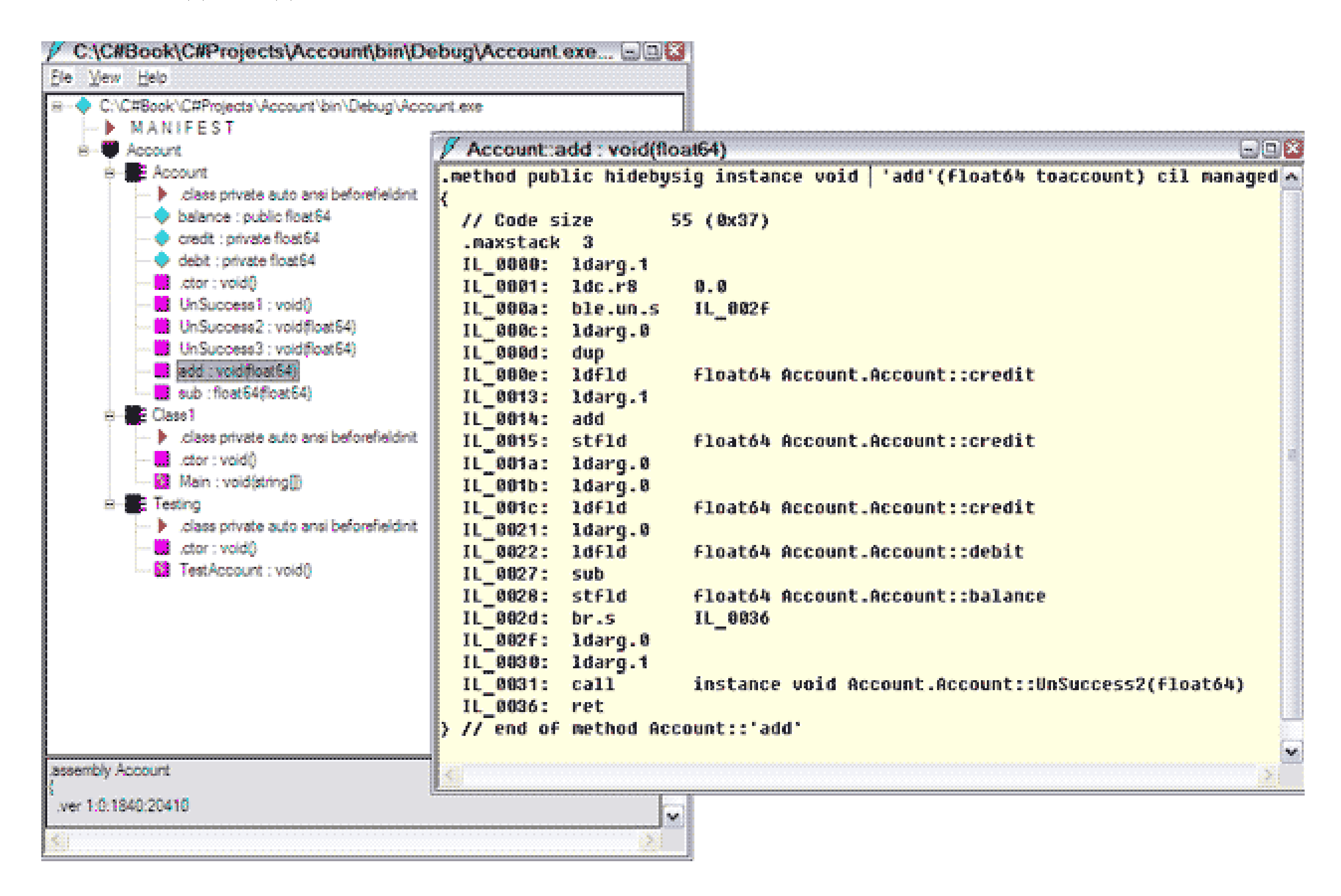


Рис. 1.1. Пример структуры PE-файла, задающего сборку

Сборщик мусора - Garbage Collector - и управление памятью

Еще одной важной особенностью построения CLR является то, что исполнительная среда берет на себя часть функций, традиционно входящих в ведение разработчиков трансляторов, и облегчает тем самым их работу. Один из таких наиболее значимых компонентов CLR - сборщик мусора (Garbage Collector). Под сборкой мусора понимается освобождение памяти, занятой объектами, которые стали бесполезными и не используются в дальнейшей работе приложения. В ряде языков программирования (классическим примером является язык C/C++) память освобождает сам программист, в явной форме отдавая команды как на создание, так и на удаление объекта. В этом есть своя логика - "я тебя породил, я тебя и убью". Однако можно и нужно освободить человека от этой работы. Неизбежные ошибки программиста при работе с памятью тяжелы по последствиям, и их крайне тяжело обнаружить. Как правило, объект удаляется в одном модуле, а необходимость в нем обнаруживается в другом, далеком модуле. Обоснование того, что программист не должен заниматься удалением объектов, а сборка мусора должна стать частью исполнительной среды, появилось достаточно давно. Наиболее полно оно обосновано в работах Бертрана Мейера и в его книге "Object-Oriented Construction Software", первое издание которой появилось еще в 1988 году.

В CLR эта идея реализована в полной мере. Задача сборки мусора снята не только с программистов, но и с разработчиков трансляторов, она решается в нужное время и в нужном месте - исполнительной средой, ответственной за выполнение вычислений. Здесь

же решаются и многие другие вопросы, связанные с использованием памяти, в частности, проверяются возможные нарушения использования "чужой" памяти и другие нарушения, например, с использованием нетипизированных указателей. Данные, удовлетворяющие требованиям CLR и допускающие сборку мусора, называются управляемыми данными.

Но, как же, спросите вы, быть с языком C++ и другими языками, где есть нетипизированные указатели, адресная арифметика, возможности удаления объектов программистом? Такие возможности сохранены и в языке C#. Ответ следующий - CLR позволяет работать как с управляемыми, так и с неуправляемыми данными. Однако использование неуправляемых данных регламентируется и не поощряется. Так, в C# модуль, использующий неуправляемые данные (указатели, адресную арифметику), должен быть помечен как небезопасный (unsafe), и эти данные должны быть четко зафиксированы. Об этом мы еще будем говорить при рассмотрении языка C# в последующих лекциях. Исполнительная среда, не ограничивая возможности языка и программистов, вводит определенную дисциплину в применении потенциально опасных средств языков программирования.

Исключительные ситуации

Что происходит, когда при вызове некоторой функции (процедуры) обнаруживается, что она не может нормальным образом выполнить свою работу? Возможны разные варианты обработки такой ситуации. Функция может возвращать код ошибки или специальное значение типа HResult, может **выбрасывать исключение**, тип которого характеризует возникшую ошибку. В CLR принято во всех таких ситуациях выбрасывать исключение. Косвенно это влияет и на язык программирования. Выбрасывание исключений наилучшим образом согласуется с исполнительной средой. В языке C# выбрасывание исключений, их дальнейший перехват и обработка - основной рекомендуемый способ обработки исключительных ситуаций.

События

У CLR есть свое видение того, что представляет собой тип. Есть формальное описание общей системы типов CTS - **Common Type System**. В соответствии с этим описанием, каждый тип, помимо полей, методов и свойств, может содержать и события. При возникновении событий в процессе работы с тем или иным объектом данного типа посылаются сообщения, которые могут получать другие объекты. Механизм обмена сообщениями основан на делегатах - функциональном типе. Надо ли говорить, что в язык C# встроен механизм событий, полностью согласованный с возможностями CLR. Мы подробно изучим все эти механизмы, рассматривая их на уровне языка.

Исполнительная среда CLR обладает мощными динамическими механизмами - сборки мусора, динамического связывания, обработки исключительных ситуаций и событий. Все эти механизмы и их реализация в CLR созданы на основании практики существующих языков программирования. Но уже созданная исполнительная среда, в свою очередь, влияет на языки, ориентированные на использование CLR. Поскольку язык C# создавался одновременно с созданием CLR, то, естественно, он стал языком, наиболее согласованным с исполнительной средой, и средства языка напрямую отображаются в средства исполнительной среды.

Общие спецификации и совместимые модули

Уже говорилось, что каркас Framework .Net облегчает межъязыковое взаимодействие. Для того чтобы классы, разработанные на разных языках, мирно уживались в рамках одного приложения, для их бесшовной отладки и возможности построения разноязычных потомков они должны удовлетворять некоторым ограничениям. Эти ограничения задаются набором

общеязыковых спецификаций - CLS (Common Language Specification). Класс, удовлетворяющий спецификациям CLS, называется CLS-совместимым. Он доступен для использования в других языках, классы которых могут быть клиентами или наследниками совместимого класса.

Спецификации CLS точно определяют, каким набором встроенных типов можно пользоваться в совместимых модулях. Понятно, что эти типы должны быть общедоступными для всех языков, использующих Framework .Net. В совместимых модулях должны использоваться управляемые данные и выполняться некоторые другие ограничения. Заметьте, ограничения касаются только интерфейсной части класса, его открытых свойств и методов. Закрытая часть класса может и не удовлетворять CLS. Классы, от которых не требуется совместимость, могут использовать специфические особенности языка программирования.

На этом я закончу обзорное рассмотрение Visual Studio .Net и ее каркаса Framework .Net. Одной из лучших книг, подробно освещающих эту тему, является книга Джеффри Рихтера, переведенная на русский язык: "Программирование на платформе .Net Framework". Крайне интересно, что для Рихтера языки являются лишь надстройкой над каркасом, поэтому он говорит о программировании, использующем возможности исполнительной среды CLR и библиотеки FCL.

2. Лекция: Язык C# и первые проекты: версия для печати и PDA
Создание языка. Его особенности. Решения, проекты, пространства имен. Консольные и Windows-приложения C#, построенные по умолчанию.

Создание C#

Язык C# является наиболее известной новинкой в области создания языков программирования. В отличие от 60-х годов XX века - периода бурного языкотворчества - в нынешнее время языки создаются крайне редко. За последние 15 лет большое влияние на теорию и практику программирования оказали лишь два языка: Eiffel, лучший, по моему мнению, объектно-ориентированный язык, и Java, ставший популярным во многом благодаря технологии его использования в Интернете и появления такого понятия как виртуальная Java-машина. Чтобы новый язык получил признание, он должен действительно обладать принципиально новыми качествами. Языку C# повезло с родителями. Явившись на свет в недрах Microsoft, будучи наследником C++, он с первых своих шагов получил мощную поддержку. Однако этого явно недостаточно для настоящего признания достоинств языка. Попробуем разобраться, имеет ли он большое будущее?

Создателем языка является сотрудник Microsoft Андреас Хейлсберг. Он стал известным в мире программистов задолго до того, как пришел в Microsoft. Хейлсберг входил в число ведущих разработчиков одной из самых популярных сред разработки - Delphi. В Microsoft он участвовал в создании версии Java - J++, так что опыта в написании языков и сред программирования ему не занимать. Как отмечал сам Андреас Хейлсберг, C# создавался как язык компонентного программирования, и в этом одно из главных достоинств языка, направленное на возможность повторного использования созданных компонентов. Из других объективных факторов отметим следующие:

- C# создавался параллельно с каркасом Framework .Net и в полной мере учитывает все его возможности - как FCL, так и CLR;
- C# является полностью объектно-ориентированным языком, где даже типы, встроенные в язык, представлены классами;
- C# является мощным объектным языком с возможностями наследования и универсализации;

- C# является наследником языков C/C++, сохраняя лучшие черты этих популярных языков программирования. Общий с этими языками синтаксис, знакомые операторы языка облегчают переход программистов от C++ к C#;
- сохранив основные черты своего великого родителя, язык стал проще и надежнее. Простота и надежность, главным образом, связаны с тем, что на C# хотя и допускаются, но не поощряются такие опасные свойства C++ как указатели, адресация, разыменование, адресная арифметика;
- благодаря каркасу Framework .Net, ставшему надстройкой над операционной системой, программисты C# получают те же преимущества работы с виртуальной машиной, что и программисты Java. Эффективность кода даже повышается, поскольку исполнительная среда CLR представляет собой компилятор промежуточного языка, в то время как виртуальная Java-машина является интерпретатором байт-кода;
- мощная библиотека каркаса поддерживает удобство построения различных типов приложений на C#, позволяя легко строить Web-службы, другие виды компонентов, достаточно просто сохранять и получать информацию из базы данных и других хранилищ данных;
- реализация, сочетающая построение надежного и эффективного кода, является немаловажным фактором, способствующим успеху C#.

Виды проектов

Как уже отмечалось, Visual Studio .Net для языков C#, Visual Basic и J# предлагает 12 возможных видов проектов. Среди них есть пустой проект, в котором изначально не содержится никакой функциональности; есть также проект, ориентированный на создание Web-служб. В этой книге, направленной, прежде всего, на изучение языка C#, основным видом используемых проектов будут обычные Windows-приложения. На начальных этапах, чтобы не усложнять задачу проблемами пользовательского интерфейса, будем рассматривать также консольные приложения.

Давайте разберемся, как создаются проекты и что они изначально собой представляют. Поговорим также о сопряженных понятиях: решение (solution), проект (project), пространство имен (namespace), сборка (assembly). Рассмотрим результаты работы компилятора Visual Studio с позиций программиста, работающего над проектом, и с позиций CLR, компилирующей PE-файл в исходный код процессора.

С точки зрения программиста, компилятор создает решение, с точки зрения CLR - сборку, содержащую PE-файл. Программист работает с решением, CLR - со сборкой.

Решение содержит один или несколько проектов, ресурсы, необходимые этим проектам, возможно, дополнительные файлы, не входящие в проекты. Один из проектов решения должен быть выделен и назначен стартовым проектом. Выполнение решения начинается со стартового проекта. Проекты одного решения могут быть зависимыми или независимыми. Например, все проекты одной лекции данной книги могут быть для удобства собраны в одном решении и иметь общие свойства . Изменяя стартовый проект, получаем возможность перехода к нужному примеру. Заметьте, стартовый проект должен иметь точку входа - класс, содержащий статическую процедуру с именем Main, которой автоматически передается управление в момент запуска решения на выполнение. В уже имеющееся решение можно добавлять как новые, так и существующие проекты. Один и тот же проект может входить в несколько решений.

Проект состоит из классов, собранных в одном или нескольких пространствах имен. Пространства имен позволяют **структурировать** проекты, содержащие большое число классов, объединяя в одну группу близкие классы. Если над проектом работает несколько

исполнителей, то, как правило, каждый из них создает свое пространство имен. Помимо структуризации, это дает возможность присваивать классам имена, не задумываясь об их уникальности. В разных пространствах имен могут существовать одноименные классы. Проект - это основная единица, с которой работает программист. Он выбирает тип проекта , а Visual Studio создает скелет проекта в соответствии с выбранным типом.

Дальнейшие объяснения лучше сочетать с реальной работой над проектами. Поэтому во всей этой книге я буду вкратце описывать свои действия по реализации тех или иных проектов, надеясь, что их повторение читателем будет способствовать пониманию текста и сути изучаемых вопросов.

Консольный проект

У себя на компьютере я открыл установленную лицензионную версию Visual Studio .Net 2003, выбрал из предложенного меню - создание нового проекта на C#, установил вид проекта - консольное приложение, дал имя проекту - ConsoleHello, указал, где будет храниться проект. Как выглядит задание этих установок, показано на рис. 2.1.

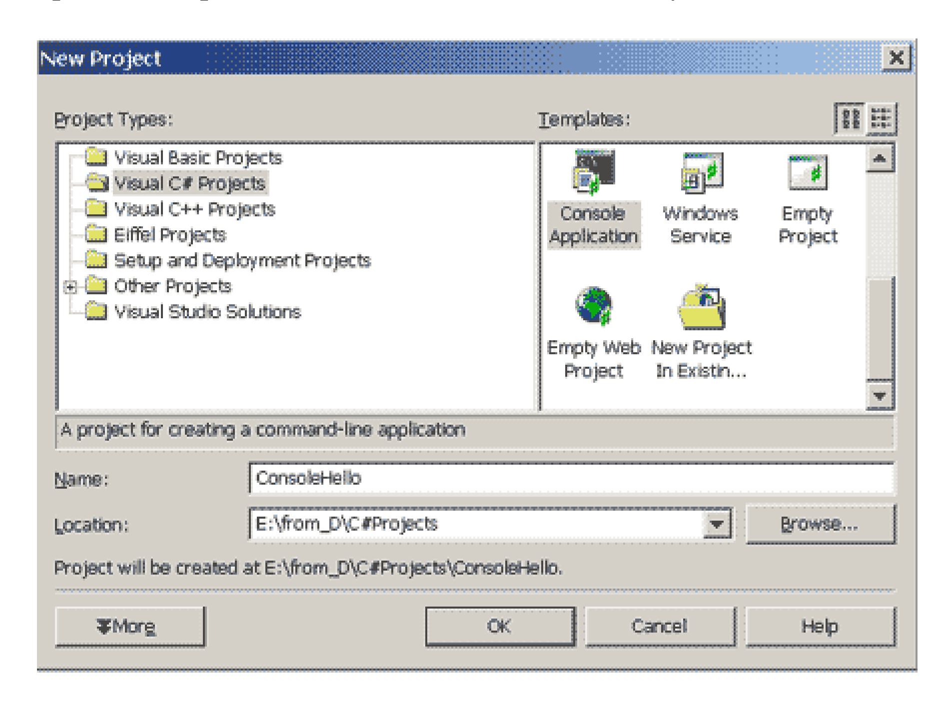


Рис. 2.1. Окно создания нового проекта

Если принять эти установки, то компилятор создаст решение, имя которого совпадает с именем проекта. На рис. 2.2 показано, как выглядит это решение в среде разработки:

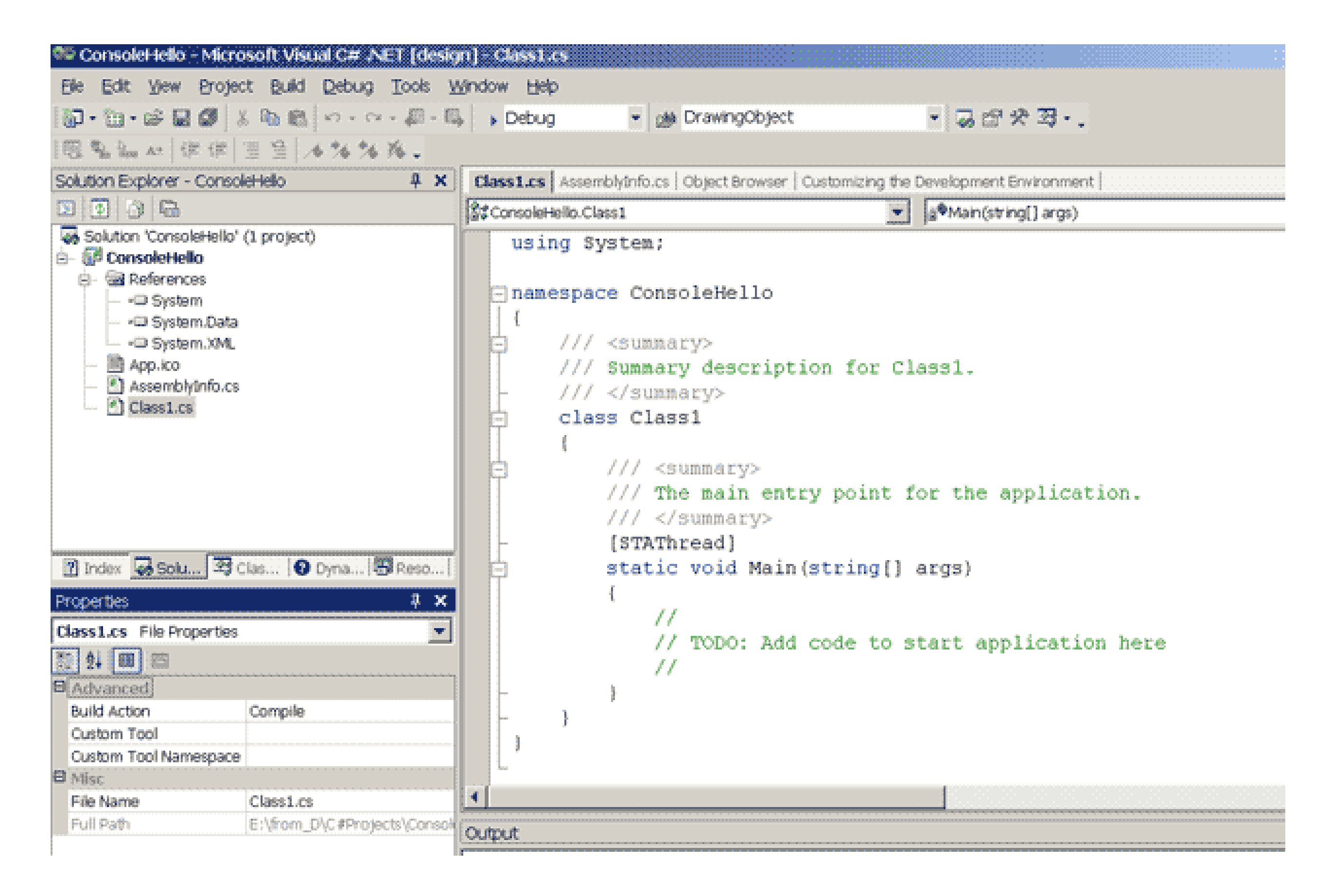


Рис. 2.2. Среда разработки и консольное приложение, построенное по умолчанию

Интегрированная среда разработки IDE (Integrated Development Environment) Visual Studio является многооконной, настраиваемой, обладает большим набором возможностей. Внешний вид ее достаточно традиционен, хотя здесь есть новые возможности; я не буду заниматься ее описанием, полагаясь на опыт читателя и справочную систему. Обращаю внимание лишь на три окна, из тех, что показаны на рис. 2.2. В окне Solution Explorer представлена структура построенного решения. В окне Properties можно увидеть свойства выбранного элемента решения. В окне документов отображается выбранный документ, в данном случае, программный код класса проекта - ConsoleHello.Class1. Заметьте, в этом окне можно отображать и другие документы, список которых показан в верхней части окна.

Построенное решение содержит, естественно, единственный заданный нами проект - ConsoleHello. Наш проект, как показано на рис. 2.2, включает в себя папку со ссылками на системные пространства имен из библиотеки FCL, файл со значком приложения и два файла с расширением cs. Файл AssemblyInfo содержит информацию, используемую в сборке, а файл со стандартным именем Class1 является построенным по умолчанию классом, который задает точку входа - процедуру Main, содержащую для данного типа проекта только комментарий.

Заметьте, класс проекта погружен в пространство имен, имеющее по умолчанию то же имя, что и решение, и проект. Итак, при создании нового проекта автоматически создается достаточно сложная вложенная структура - решение, содержащее проект, содержащий пространство имен, содержащее класс, содержащий точку входа. Для простых решений такая структурированность представляется избыточной, но для сложных - она осмысленна и полезна.

Помимо понимания структуры решения, стоит также разобраться в трех важных элементах, включенных в начальный проект - предложение using, тэги документации в комментариях и атрибуты.

Пространству имен может предшествовать одно или несколько предложений using, где после ключевого слова следует название пространства имен - из библиотеки FCL или из проектов, связанных с текущим проектом. В данном случае задается пространство имен System - основное пространство имен библиотеки FCL. Предложение using NameA облегчает запись при использовании классов, входящих в пространство NameA, поскольку в этом случае не требуется каждый раз задавать полное имя класса с указанием имени пространства, содержащего этот класс. Чуть позже мы увидим это на примере работы с классом Console пространства System. Заметьте, полное имя может потребоваться, если в нескольких используемых пространствах имен имеются классы с одинаковыми именами.

Все языки допускают комментарии. В C#, как и в C++, допускаются однострочные и многострочные комментарии. Первые начинаются с двух символов косой черты. Весь текст до конца строки, следующий за этой парой символов, (например, "//любой текст") воспринимается как комментарий, не влияющий на выполнение программного кода. Началом многострочного комментария является пара символов /*, а концом - */. Заметьте, тело процедуры Main содержит три однострочных комментария.

Здесь же, в проекте, построенном по умолчанию, мы встречаемся с еще одной весьма важной новинкой C# - **XML-тегами**, формально являющимися частью комментария. Отметим, что описанию класса Class1 и описанию метода Main предшествует заданный в строчном комментарии XML-тег <summary>. Этот тэг распознается специальным инструментарием, строящим XML-отчет проекта. Идея самодокументируемых программных проектов, у которых документация является неотъемлемой частью, является важной составляющей стиля компонентного надежного программирования на C#. Мы рассмотрим реализацию этой идеи в свое время более подробно, но уже с первых шагов будем использовать теги документирования и строить XML-отчеты. Заметьте, кроме тега <summary> возможны и другие тэги, включаемые в отчеты. Некоторые теги добавляются почти автоматически. Если в нужном месте (перед объявлением класса, метода) набрать подряд три символа косой черты, то автоматически вставится тэг документирования, так что останется только дополнить его соответствующей информацией.

Еще одна новинка C#, встречающаяся в начальном проекте, это атрибут [STAThread], который предшествует описанию процедуры Main. Так же, как и тэги документирования, атрибуты распознаются специальным инструментарием и становятся частью метаданных. Атрибуты могут быть как стандартными, так и заданными пользователем. Стандартные атрибуты используются CLR и влияют на то, как она будет выполнять проект. В данном случае атрибут [STAThread] (Single Thread Apartment) задает однопоточную модель выполнения. Об атрибутах и метаданных мы еще будем говорить подробно. Заметьте, если вы нечетко представляете, каков смысл однопоточной модели, и не хотите, чтобы в вашем тексте присутствовали непонятные для вас указания, то этот атрибут можно удалить из текста, что не отразится на выполнении.

Скажем еще несколько слов о точке входа - процедуре Main. Ее заголовок можно безболезненно упростить, удалив аргументы, которые, как правило, не задаются. Они имеют смысл, когда проект вызывается из командной строки, позволяя с помощью параметров задать нужную стратегию выполнения проекта.

Таков консольный проект, построенный по умолчанию. Функциональности у него немного. Его можно скомпилировать, выбрав соответствующий пункт из меню **build**. Если компиляция прошла без ошибок, то в результате будет произведена сборка и появится PE-файл в соответствующей папке Debug нашего проекта. Приложение можно запустить нажатием соответствующих клавиш (например, CTRL+F5) или выбором соответствующего пункта из меню **Debug**. Приложение будет выполнено под управлением CLR. В результате выполнения появится консольное окно с предложением нажать любую клавишу для

закрытия окна.

Слегка изменим проект, построенный по умолчанию, добавим в него свой код и превратим его в классический проект приветствия. Нам понадобятся средства для работы с консолью - чтения с консоли (клавиатуры) и вывода на консоль (дисплей) строки текста. Библиотека FCL предоставляет для этих целей класс Console, среди многочисленных методов которого есть методы ReadLine и WriteLine с очевидной семантикой. Вот код проекта, полученный в результате моих корректировок:

```
using System;
namespace ConsoleHello
{
	/// <summary>
	/// Первый консольный проект - Приветствие
	/// </summary>
	class Class1
	{
		/// <summary>
		/// Точка входа. Запрашивает имя и выдает приветствие
		/// </summary>
		static void Main()
		{
			Console.WriteLine("Введите Ваше имя");
			string name;
			name = Console.ReadLine();
			if (name=="")
				Console.WriteLine ("Здравствуй, мир!");
			else
				Console.WriteLine("Здравствуй, " + name + "!");
		}
	}
}
```

Я изменил текст в тегах <summary>, удалил атрибут и аргументы процедуры Main, добавил в ее тело операторы ввода-вывода. Благодаря предложению using, мне не требуется при вызове методов класса Console каждый раз писать System.Console. Надеюсь, что программный текст понятен без дальнейших пояснений.

В завершение первого проекта построим его XML-отчет. Для этого в свойствах проекта необходимо указать имя файла, в котором будет храниться отчет. Установка этого свойства проекта, так же как и других свойств, делается в окне Property Pages, открыть которое можно по-разному. Я обычно делаю это так: в окне Solution Explorer выделяю строку с именем проекта, а затем в окне Properties нажимаю имеющуюся там кнопку Property Pages. Затем в открывшемся окне свойств, показанном на рис. 2.3, устанавливается нужное свойство. В данном случае я задал имя файла отчета hello.xml.

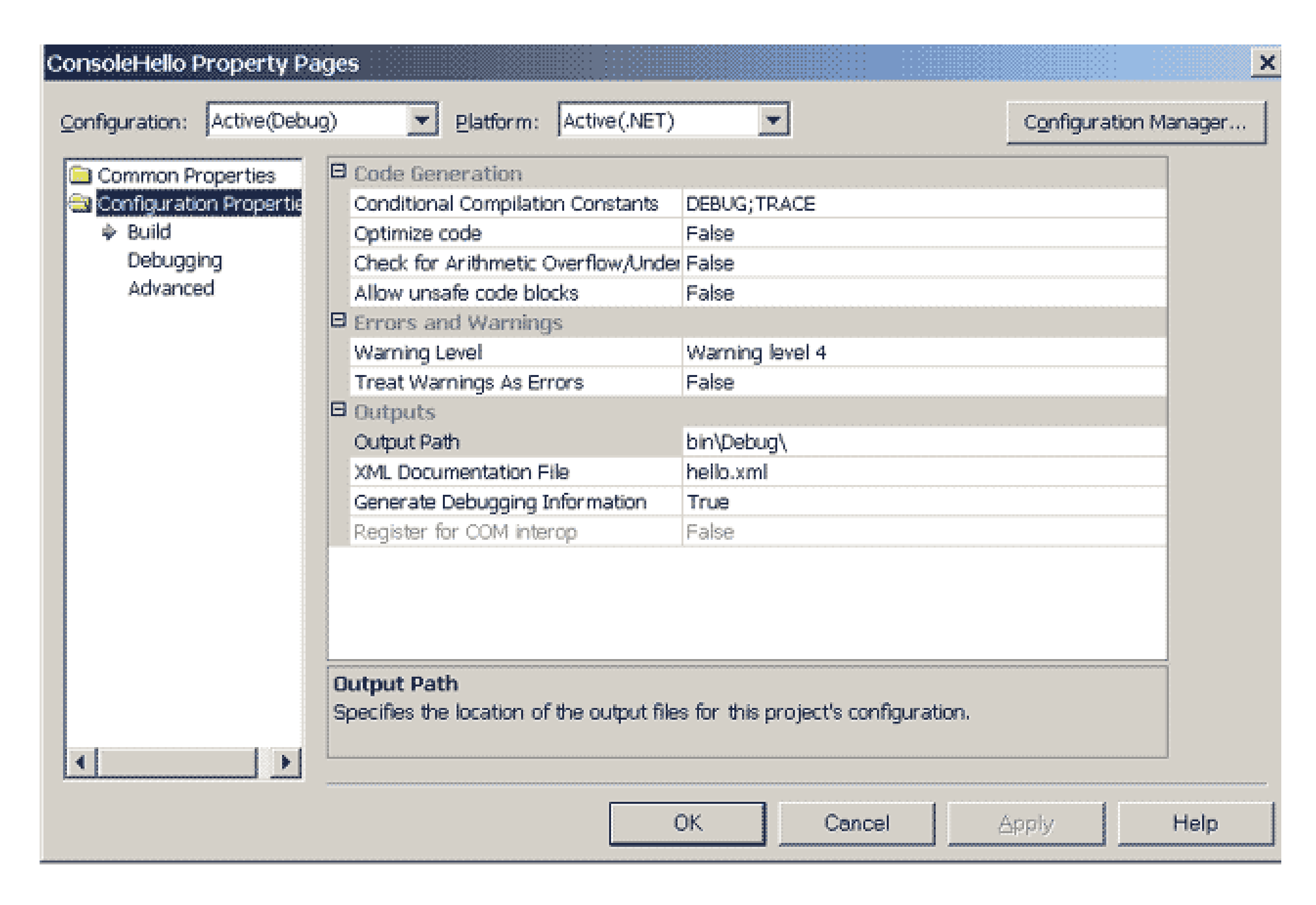


Рис. 2.3. Окно Property Pages проекта и задание имени XML-отчета

После перестройки проекта можно открыть этот файл с документацией. Если соблюдать дисциплину, то в нем будет задана спецификация проекта, с описанием всех классов, их свойств и методов. Вот как выглядит этот отчет в данном примере:

```
<?xml version="1.0"?>
<doc>
<assembly>
        <name>ConsoleHello</name>
</assembly>
<members>
        <member name="T:ConsoleHello.Class1">
                <summary>
                        Первый консольный проект - Приветствие
                </summary>
        </member>
        <member name="M:ConsoleHello.Class1.Main">
                <summary>
                        Точка входа. Запрашивает имя и_выдает приветствие
                </summary>
        </member>
</members>
</doc>
```

Как видите, отчет описывает наш проект, точнее, сборку. Пользователь, пожелавший воспользоваться этой сборкой, из отчета поймет, что она содержит один класс, назначение которого указано в теге <summary>. Класс содержит лишь один элемент - точку входа Main с заданной спецификацией в теге <summary>.

Windows-проект

Проделаем аналогичную работу: построим Windows- проект, рассмотрим, как он выглядит по умолчанию, а затем дополним его до проекта "Приветствие". Повторяя уже описанные

действия, в окне нового проекта (см. рис. 2.1) я выбрал тип проекта Windows Application, дав проекту имя WindowsHello.

Как и в консольном случае, по умолчанию строится решение, содержащее единственный проект, содержащий единственное пространство имен (все три конструкции имеют совпадающие имена). В пространство имен вложен единственный класс Form1, но это уже далеко не столь простой класс, как ранее. Вначале приведу его код, а потом уже дам необходимые пояснения:

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;
namespace WindowsHello
{
	/// <summary>
	/// Summary description for Form1.
	/// </summary>
	public class Form1 : System.Windows.Forms.Form
	{
		/// <summary>
		/// Required designer variable.
		/// </summary>
		private System.ComponentModel.Container components = null;
		public Form1()
		{
			// Required for Windows Form Designer support
			InitializeComponent();
			// TODO: Add any constructor code after
			// InitializeComponent call
		}
		/// <summary>
		/// Clean up any resources being used.
		/// </summary>
		protected override void Dispose( bool disposing )
		{
			if( disposing )
			{
				if (components != null)
				{
					components.Dispose();
				}
			}
			base.Dispose( disposing );
		}
		#region Windows Form Designer generated code
		/// <summary>
		/// Required method for Designer support - do not modify
		/// the contents of this method with the code editor.
		/// </summary>
		private void InitializeComponent()
		{
			this.components = new
				System.ComponentModel.Container();
			this.Size = new System.Drawing.Size(300,300);
			this.Text = "Form1";
		}
		#endregion
		/// <summary>
		/// The main entry point for the application.
		/// </summary>
		[STAThread]
```

```
                static void Main()
                {
                        Application.Run(new Form1());
                }
        }
}
```

Начну с того, что теперь пространству имен предшествует 6 предложений using; это означает, что используются не менее 6-ти классов, находящихся в разных пространствах имен библиотеки FCL. Одним из таких используемых классов является класс Form из глубоко вложенного пространства имен System.Windows.Forms. Построенный по умолчанию класс Form1 является наследником класса Form и автоматически наследует его функциональность - свойства, методы, события. При создании объекта этого класса, характеризующего форму, одновременно Visual Studio создает визуальный образ объекта - окно, которое можно заселять элементами управления. В режиме проектирования эти операции можно выполнять вручную, при этом автоматически происходит изменение программного кода класса. Появление в проекте формы, открывающейся по умолчанию при запуске проекта, означает переход к визуальному, управляемому событиями программированию. Сегодня такой стиль является общепризнанным, а стиль консольного приложения следует считать анахронизмом, правда, весьма полезным при изучении свойств языка.

В класс Form1 встроено закрытое (private) свойство - объект components класса Container. В классе есть конструктор, вызывающий закрытый метод класса InitializeComponent. В классе есть деструктор, освобождающий занятые ресурсы, которые могут появляться при добавлении элементов в контейнер components. Наконец, в классе есть точка входа - процедура Main с непустым телом.

Начало начал - точка "большого взрыва"

Основной операцией, инициирующей вычисления в объектно-ориентированных приложениях, является вызов метода F некоторого класса x, имеющий вид:

```
x.F(arg1, arg2, ..., argN);
```

В этом вызове x называется целью вызова, и здесь возможны три ситуации:

- x - имя класса. В этом случае метод F должен быть статическим методом класса, объявленным с атрибутом static, как это имеет место, например, для точки вызова - процедуры Main ;
- x - имя объекта или объектное выражение. В этом случае F должен быть обычным, не статическим методом. Иногда такой метод называют экземплярным, подчеркивая тот факт, что метод вызывается экземпляром класса - некоторым объектом;
- x - не указывается при вызове. Такой вызов называется неквалифицированным, в отличие от двух первых случаев. Заметьте, неквалифицированный вызов вовсе не означает, что цель вызова отсутствует, - она просто задана по умолчанию. Целью является текущий объект (текущий класс для статических методов). Текущий объект имеет зарезервированное имя this. Применяя это имя, любой неквалифицированный вызов можно превратить в квалифицированный вызов. Иногда без этого имени просто не обойтись.

Но как появляются объекты? Как они становятся текущими? Как реализуется самый первый вызов метода, другими словами, кто и где вызывает точку входа - метод Main? С чего все начинается?

Когда CLR получает сборку для выполнения, то в решении, входящем в сборку, отмечен стартовый проект, содержащий класс с точкой входа - статическим методом (процедурой) Main. Некоторый объект исполнительной среды CLR и вызывает этот метод, так что первоначальный вызов метода осуществляется извне приложения. Это и есть точка "большого взрыва" - начало зарождения мира объектов и объектных вычислений. Дальнейший сценарий зависит от содержимого точки входа. Как правило, в ней создаются один или несколько объектов, а затем вызываются методы и/или обработчики событий, происходящих с созданными объектами. В этих методах и обработчиках событий могут создаваться новые объекты, вызываться новые методы и новые обработчики. Так, начиная с одной точки, разворачивается целый мир объектов приложения.

Выполнение проекта по умолчанию после "большого взрыва"

Давайте посмотрим, что происходит в проекте, создаваемом по умолчанию, когда произошел "большой взрыв", вселенная создана и процедура Main начала работать. Процедура Main содержит всего одну строчку:

```
Application.Run(new Form1());
```

Прокомментируем этот квалифицированный вызов. Целью здесь является класс Application из пространства имен System.Windows.Forms. Класс вызывает статический метод Run, которому в качестве фактического аргумента передается объектное выражение new Form1(). При вычислении этого выражения создается первый объект - экземпляр класса Form1. Этот объект становится текущим. Для создания объекта вызывается конструктор класса. В процессе работы конструктора осуществляется неквалифицированный вызов метода InitializeComponent(). Целью этого вызова является текущий объект - уже созданный объект класса Form1. Ни в конструкторе, ни в вызванном методе новые объекты не создаются. По завершении работы конструктора объект класса Form1 передается методу Run в качестве аргумента.

Метод Run класса Application - это знаменитый метод. Во-первых, он открывает форму - видимый образ объекта класса Form1, с которой теперь может работать пользователь. Но главная его работа состоит в том, что он создает настоящее Windows-приложение, запуская цикл обработки сообщений о происходящих событиях. Поступающие сообщения обрабатываются операционной системой согласно очереди и приоритетам, вызывая обработчики соответствующих событий. Поскольку наша форма по умолчанию не заселена никакими элементами управления, то поступающих сообщений немного. Все, что может делать пользователь с формой, так это перетаскивать ее по экрану, сворачивать и изменять размеры. Конечно, он может еще закрыть форму. Это приведет к завершению цикла обработки сообщений, к завершению работы метода Run, к завершению работы метода Main, к завершению работы приложения.

Проект WindowsHello

Давайте расширим приложение по умолчанию до традиционного приветствия в Windows-стиле, добавив окошки для ввода и вывода информации. Как уже говорилось, при создании Windows-приложения по умолчанию создается не только объект класса Form1 - потомка класса Form, но и его видимый образ - форма, с которой можно работать в режиме проектирования, населяя ее элементами управления. Добавим в форму следующие элементы управления:

- текстовое окно и метку. По умолчанию они получат имена textBox1 и label1. Текстовое окно предназначается для ввода имени пользователя, метка, визуально связанная с окном, позволит указать назначение текстового окна. Я установил

свойство Multiline для текстового окна как true, свойство Text у метки - Ваше Имя;

- аналогичная пара элементов управления - textBox2 и label2 - предназначены для вывода приветствия. Поскольку окно textBox2 предназначено для вывода, то я включил его свойство ReadOnly;
- я посадил на форму командную кнопку, обработчик события Click которой и будет организовывать чтение имени пользователя из окна textBox1 и вывод приветствия в окно textBox2.

На рис. 2.4 показано, как выглядит наша форма в результате проектирования.

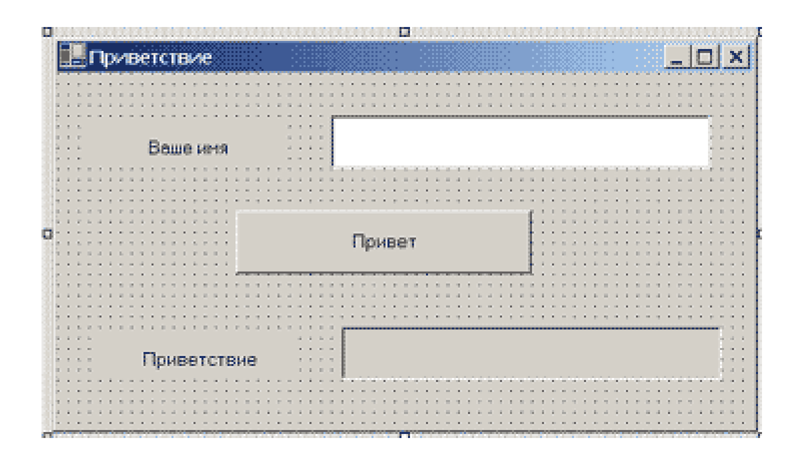


Рис. 2.4. Форма "Приветствие"

Я не буду далее столь же подробно описывать действия по проектированию интерфейса форм, полагая, что все это интуитивно ясно и большинству хорошо знакомо. Более важно понимать то, что все действия по проектированию интерфейса незамедлительно транслируются в программный код, добавляемый в класс **Form1**. Мы вручную сажаем элемент управления на форму, тут же в классе появляется закрытое свойство, задающее этот элемент, а в процедуре **InitailizeComponent** выполняется его инициализация. Мы меняем некоторое свойство элемента управления, это незамедлительно находит отражение в программном коде указанной процедуры.

Вот как выглядит автоматически добавленное в класс описание элементов управления:

```
private System.Windows.Forms.Label label1;
        private System.Windows.Forms.TextBox textBox1;
        private System.Windows.Forms.Button button1;
        private System.Windows.Forms.TextBox textBox2;
        private System.Windows.Forms.Label label2;
```

А вот фрагмент текста процедуры InitailizeComponent:

```
#region Windows Form Designer generated code
        /// <summary>
        /// Required method for Designer support - do not
        /// modify the contents of this method with the code
        /// editor.
        /// </summary>
        private void InitializeComponent()
        {
                this.label1 = new System.Windows.Forms.Label();
                this.textBox1 = new System.Windows.Forms.TextBox();
                this.button1 = new System.Windows.Forms.Button();
                this.textBox2 = new System.Windows.Forms.TextBox();
                this.label2 = new System.Windows.Forms.Label();
                this.SuspendLayout();
                // label1
```

```
                this.label1.Location = new System.Drawing.Point(24, 40);
                this.label1.Name = "label1";
                this.label1.Size = new System.Drawing.Size(152, 32);
                this.label1.TabIndex = 0;
                this.label1.Text = "Ваше имя";
                this.label1.TextAlign =
System.Drawing.ContentAlignment.MiddleCenter;
... аналогично задаются описания свойств всех элементов управления ...
... далее задаются свойства самой формы ...
                        // Form1
                        //
                        this.AutoScaleBaseSize = new System.Drawing.Size(6, 15);
                        this.ClientSize = new System.Drawing.Size(528, 268);
                        this.Controls.AddRange(new
                                System.Windows.Forms.Control[]
                                {
                                        this.textBox2,
                                        this.label2,
                                        this.button1,
                                        this.textBox1,
                                        this.label1
                                });
                        this.Name = "Form1";
                        this.Text = "Приветствие";
                        this.Load += new System.EventHandler(this.Form1_Load);
                        this.ResumeLayout(false);
                }
                #endregion
```

Заметьте, в теге <summary> нас предупреждают, что этот метод требуется специальному инструментарию - Дизайнеру формы - и он не предназначен для редактирования пользователем; добавление и удаление кода этого метода производится автоматически. Обращаю внимание, что после заполнения свойств элементов управления заключительным шагом является их добавление в коллекцию Controls, хранящую все элементы управления. Здесь используется метод AddRange, позволяющий добавить в коллекцию одним махом целый массив элементов управления. Метод Add позволяет добавлять в коллекцию по одному элементу. Позже нам придется добавлять элементы управления в форму программно, динамически изменяя интерфейс формы. Для этого мы будем выполнять те же операции: объявить элемент управления, создать его, используя конструкцию new, задать нужные свойства и добавить в коллекцию Controls.

В заключение приведу текст обработчика событий командной кнопки. Как задается обработчик того или иного события для элементов управления? Это можно делать по-разному. Есть стандартный способ включения событий. Достаточно выделить нужный элемент в форме, в окне свойств нажать кнопку событий (со значком молнии) и из списка событий выбрать нужное событие и щелкнуть по нему. В данной ситуации все можно сделать проще - двойной щелчок по кнопке включает событие, и автоматически строится заготовка обработчика события с нужным именем и параметрами. Вот как она выглядит:

```
private void button1_Click(object sender,System.EventArgs e)
        {

        }
```

Нам остается добавить свой текст. Я добавил следующие строки:

```
string temp;
        temp = textBox1.Text;
                if( temp == "")
                        textBox2.Text = "Здравствуй, мир!";
                else
```

```
textBox2.Text = "Здравствуй, " + temp + " !";
```

И вот как это работает.

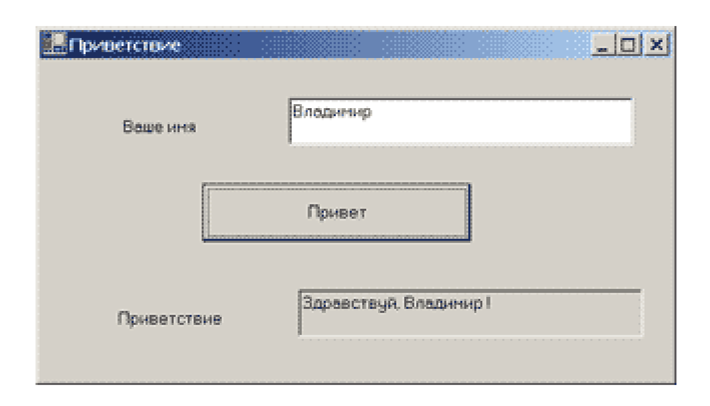


Рис. 2.5. Форма "Приветствие" в процессе работы

На этом мы закончим первое знакомство с проектами на C# и в последующих лекциях приступим к систематическому изучению возможностей языка.

3. Лекция: Система типов языка C#: версия для печати и PDA
Общий взгляд. Система типов. Типы-значения и ссылочные типы. Встроенные типы. Сравнение с типами C++. Типы или классы? И типы, и классы! Преобразования переменных в объекты и vice versa. Операции "упаковать" и "распаковать". Преобразования типов. Преобразования внутри арифметического типа. Преобразования строкового типа. Класс Convert и его методы. Проверяемые преобразования. Управление проверкой арифметических преобразований.

Общий взгляд

Знакомство с новым языком программирования разумно начинать с изучения системы типов этого языка. Как в нем устроена система типов данных? Какие есть простые типы, как создаются сложные, структурные типы, как определяются собственные типы, динамические типы, как определяются классы?

В первых языках программирования понятие класса отсутствовало - рассматривались только типы данных. При определении типа явно задавалось только множество возможных значений, которые могут принимать переменные этого типа. Например, тип integer задает целые числа в некотором диапазоне. Неявно с типом всегда связывался и набор разрешенных операций. В типизированных языках, к которым относится большинство языков программирования, понятие переменной естественным образом связывалось с типом. Если есть тип T и переменная x типа T, то это означало, что переменная может принимать значения из множества, заданного типом, и к ней применимы операции, разрешенные типом.

Классы и объекты впервые появились в программировании в языке Симула 67. Произошло это спустя 10 лет после появления первого алгоритмического языка Фортран. Определение класса наряду с описанием данных содержало четкое определение операций или методов, применимых к данным. Объекты - экземпляры класса являются обобщением понятия переменной. Сегодня определение класса в C# и других объектных языках, аналогично определению типа в CTS, содержит:

- данные, задающие свойства объектов класса ;
- методы, определяющие поведение объектов класса ;
- события, которые могут происходить с объектами класса.

Так есть ли различие между этими двумя основополагающими понятиями - типом и классом? На первых порах можно считать, что класс - это хорошо определенный тип данных, объект - хорошо определенная переменная. Понятия фактически являются синонимами, какое из них употреблять лишь дело вкуса. Встроенные типы, такие как integer или string, предпочитают называть по-прежнему типами, а их экземпляры - переменными. Что же касается абстракции данных, описывающей служащих и названной, например, Employee, то естественнее называть ее классом, а ее экземпляры - объектами. Такой взгляд на типы и классы довольно полезен, но он не является полным. Позже при обсуждении классов и наследования постараемся более четко определить принципиальные различия в этих понятиях.

Объектно-ориентированное программирование, доминирующее сегодня, построено на классах и объектах. Тем не менее, понятия типа и переменной все еще остаются центральными при описании языков программирования, что характерно и для языка C#. Заметьте, что и в Framework .Net предпочитают говорить о системе типов, хотя все типы библиотеки FCL являются классами.

Типы данных принято разделять на простые и сложные в зависимости от того, как устроены их данные. У простых (скалярных) типов возможные значения данных едины и неделимы. Сложные типы характеризуются способом структуризации данных - одно значение сложного типа состоит из множества значений данных, организующих сложный тип.

Есть и другие критерии классификации типов. Так, типы разделяются на встроенные типы и типы, определенные программистом (пользователем). Встроенные типы изначально принадлежат языку программирования и составляют его базис. В основе системы типов любого языка программирования всегда лежит базисная система типов, встроенных в язык. На их основе программист может строить собственные, им самим определенные типы данных. Но способы (правила) создания таких типов являются базисными, встроенными в язык.

Типы данных разделяются также на статические и динамические. Для данных статического типа память отводится в момент объявления, требуемый размер данных (памяти) известен при их объявлении. Для данных динамического типа размер данных в момент объявления обычно неизвестен и память им выделяется динамически по запросу в процессе выполнения программы.

Еще одна важная классификация типов - это их деление на значимые и ссылочные. Для значимых типов значение переменной (объекта) является неотъемлемой собственностью переменной (точнее, собственностью является память, отводимая значению, а само значение может изменяться). Для ссылочных типов значением служит ссылка на некоторый объект в памяти, расположенный обычно в динамической памяти - "куче". Объект, на который указывает ссылка, может быть разделяемым. Это означает, что несколько ссылочных переменных могут указывать на один и тот же объект и разделять его значения. Значимый тип принято называть развернутым, подчеркивая тем самым, что значение объекта развернуто непосредственно в памяти, отводимой объекту. О ссылочных и значимых типах еще предстоит обстоятельный разговор.

Для большинства процедурных языков, реально используемых программистами - Паскаль, C++, Java, Visual Basic, C#, - система встроенных типов более или менее одинакова. Всегда в языке присутствуют арифметический, логический (булев), символьный типы. Арифметический тип всегда разбивается на подтипы. Всегда допускается организация данных в виде массивов и записей (структур). Внутри арифметического типа всегда допускаются **преобразования**, всегда есть функции, преобразующие строку в число и обратно. Так что, мой читатель, Ваше знание, по крайней мере, одного из процедурных

языков позволяет построить общую картину системы типов и для языка C#. Отличия будут в нюансах, которые и придают аромат и неповторимость языку.

Поскольку язык C# является непосредственным потомком языка C++, то и системы типов этих двух языков близки и совпадают вплоть до названия типов и областей их определения. Но отличия, в том числе принципиального характера, есть и здесь.

Система типов

Давайте рассмотрим, как устроена система типов в языке C#, но вначале для сравнения приведу классификацию типов в стандарте языка C++.

Стандарт языка C++ включает следующий набор **фундаментальных** типов.

1. Логический тип (bool).
2. Символьный тип (char).
3. Целые типы. Целые типы могут быть одного из трех размеров - short, int, long, сопровождаемые описателем signed или unsigned, который указывает, как интерпретируется значение, - со знаком или без оного.
4. Типы с плавающей точкой. Эти типы также могут быть одного из трех размеров - float, double, long double. Кроме того, в языке есть тип void, используемый для указания на отсутствие информации. Язык позволяет конструировать типы.
5. Указатели (например, int* - типизированный указатель на переменную типа int).
6. Ссылки (например, double& - типизированная ссылка на переменную типа double).
7. Массивы (например, char[] - массив элементов типа char).

   Язык позволяет конструировать пользовательские типы

8. **Перечислимые** типы (enum) для представления значений из конкретного множества.
9. Структуры (struct).
10. Классы.

Первые три вида типов называются интегральными или счетными. Значения их перечислимы и упорядочены. Целые типы и типы с плавающей точкой относятся к арифметическому типу. Типы подразделяются также на встроенные и типы, определенные пользователем.

Эта схема типов сохранена и в языке C#. Однако здесь на верхнем уровне используется и другая классификация, носящая для C# принципиальный характер. Согласно этой классификации все типы можно разделить на четыре категории:

1. Типы-значения (value), или значимые типы.
2. Ссылочные (reference).
3. Указатели (pointer).
4. Тип void.

Эта классификация основана на том, где и как хранятся значения типов. Для ссылочного типа значение задает ссылку на область памяти в "куче", где расположен соответствующий объект. Для значимого типа используется прямая адресация, значение хранит собственно данные, и память для них отводится, как правило, в стеке.

В отдельную категорию выделены указатели, что подчеркивает их особую роль в языке. Указатели имеют ограниченную область действия и могут использоваться только в

небезопасных блоках, помеченных как unsafe.

Особый статус имеет и тип void, указывающий на отсутствие какого-либо значения.

В языке C# жестко определено, какие типы относятся к ссылочным, а какие - к значимым. К значимым типам относятся: логический, арифметический, структуры, перечисление. Массивы, **строки** и классы относятся к ссылочным типам. На первый взгляд, такая классификация может вызывать некоторое недоумение, почему это структуры, которые в C++ близки к классам, относятся к значимым типам, а массивы и строки - к ссылочным. Однако ничего удивительного здесь нет. В C# массивы рассматриваются как динамические, их размер может определяться на этапе вычислений, а не в момент трансляции. Строки в C# также рассматриваются как динамические переменные, длина которых может изменяться. Поэтому строки и массивы относятся к ссылочным типам, требующим распределения памяти в "куче".

Со структурами дело сложнее. Структуры C# представляют частный случай класса. Определив свой класс как структуру, программист получает возможность отнести класс к значимым типам, что иногда бывает крайне полезно. Замечу, что в хорошем объектном языке Eiffel программист может любой класс объявить развернутым (expanded), что эквивалентно отнесению к значимому типу. У программиста C# только благодаря структурам появляется возможность управлять отнесением класса к значимым или ссылочным типам. Правда, это неполноценное средство, поскольку на структуры накладываются дополнительные ограничения по сравнению с обычными классами.

Рассмотрим классификацию, согласно которой все типы делятся на встроенные и определенные пользователем. Все встроенные типы C# однозначно отображаются, и фактически совпадают с системными типами каркаса Net Framework, размещенными в пространстве имен System. Поэтому всюду, где можно использовать имя типа, например, - int, с тем же успехом можно использовать и имя System.Int32.

Замечание:

Следует понимать тесную связь и идентичность встроенных типов языка C# и типов каркаса. Какими именами типов следует пользоваться в программных текстах - это спорный вопрос. Джеффри Рихтер в своей известной книге "Программирование на платформе Framework .Net" рекомендует использовать системные имена. Другие авторы считают, что следует пользоваться именами типов, принятыми в языке. Возможно, в модулях, предназначенных для межъязыкового взаимодействия, разумны системные имена, а в остальных случаях - имена конкретного языка программирования.

В заключение этого раздела приведу таблицу (3.1), содержащую описание всех встроенных типов языка C# и их основные характеристики.

Логический тип			
Имя типа	**Системный тип**	**Значения**	**Размер**
Bool	System.Boolean	true, false	8 бит
Арифметические целочисленные типы			
Имя типа	**Системный тип**	**Диапазон**	**Размер**
Sbyte	System.SByte	-128 — 127	Знаковое, 8 Бит
Byte	System.Byte	0 — 255	Беззнаковое, 8 Бит
Short	System.Short	-32768 —32767	Знаковое, 16 Бит

Ushort	System.UShort	0 — 65535	Беззнаковое, 16 Бит
Int	System.Int32	≈ (-2*10^9 — 2*10^9)	Знаковое, 32 Бит
Uint	System.UInt32	≈ (0 — 4*10^9)	Беззнаковое, 32 Бит
Long	System.Int64	≈ (-9*10^18 — 9*10^18)	Знаковое, 64 Бит
Ulong	System.UInt64	≈ (0— 18*10^18)	Беззнаковое, 64 Бит
Арифметический тип с плавающей точкой			
Имя типа	**Системный тип**	**Диапазон**	**Точность**
Float	System.Single	+1.5*10^-45 -/+3.4*10^38	7 цифр
Double	System.Double	+5.0*10^-324 -/+1.7*10^308	15-16 цифр
Арифметический тип с фиксированной точкой			
Имя типа	**Системный тип**	**Диапазон**	**Точность**
Decimal	System.Decimal	+1.0*10^-28 - +7.9*10^28	28-29 значащих цифр
Символьные типы			
Имя типа	**Системный тип**	**Диапазон**	**Точность**
Char	System.Char	U+0000 - U+ffff	16 бит Unicode символ
String	System.String	Строка из символов Unicode	
Объектный тип			
Имя типа	**Системный тип**	**Примечание**	
Object	System.Object	Прародитель всех встроенных и пользовательских типов	

Система встроенных типов языка C# не только содержит практически все встроенные типы (за исключением long double) стандарта языка C++, но и перекрывает его разумным образом. В частности тип string является встроенным в язык, что вполне естественно. В области совпадения сохранены имена типов, принятые в C++, что облегчает жизнь тем, кто привык работать на C++, но собирается по тем или иным причинам перейти на язык C#.

Типы или классы? И типы, и классы

Язык C# в большей степени, чем язык C++, является языком объектного программирования. В чем это выражается? В языке C# сглажено различие между типом и классом. Все типы - встроенные и пользовательские - одновременно являются классами, связанными отношением наследования. Родительским, базовым классом является класс Object. Все остальные типы или, точнее, классы являются его потомками, наследуя методы этого класса. У класса Object есть четыре наследуемых метода:

1. bool Equals (object obj) - проверяет эквивалентность текущего объекта и объекта, переданного в качестве аргумента;
2. System.Type GetType () - возвращает системный тип текущего объекта;
3. string ToString () - возвращает строку, связанную с объектом. Для арифметических типов возвращается значение, преобразованное в строку;
4. int GetHashCode() - служит как хэш-функция в соответствующих алгоритмах поиска по ключу при хранении данных в хэш-таблицах.

Естественно, что все встроенные типы нужным образом переопределяют методы родителя и добавляют собственные методы и свойства. Учитывая, что и типы, создаваемые пользователем, также являются потомками класса Object, то для них необходимо переопределить методы родителя, если предполагается использование этих методов ; реализация родителя, предоставляемая по умолчанию, не обеспечивает нужного эффекта.

Перейдем теперь к примерам, на которых будем объяснять дальнейшие вопросы, связанные с типами и классами, переменными и объектами. Начнем с вполне корректного в языке C# примера объявления переменных и присваивания им значений:

```
int x=11;
int v = new Int32();
v = 007;
string s1 = "Agent";
s1 = s1 + v.ToString() +x. ToString();
```

В этом примере переменная x объявляется как обычная переменная типа int. В то же время для объявления переменной v того же типа int используется стиль, принятый для объектов. В объявлении применяется конструкция new и вызов конструктора класса. В операторе присваивания, записанном в последней строке фрагмента, для обеих переменных вызывается метод ToString, как это делается при работе с объектами. Этот метод, наследуемый от родительского класса Object, переопределенный в классе int, возвращает строку с записью целого. Сообщу еще, что класс int не только наследует методы родителя - класса Object, - но и дополнительно определяет метод CompareTo, выполняющий сравнение целых, и метод GetTypeCode, возвращающий системный код типа. Для класса Int определены также статические методы и поля, о которых расскажу чуть позже.

Так что же такое после этого int, спросите Вы: тип или класс? Ведь ранее говорилось, что int относится к value-типам, следовательно, он хранит в стеке значения своих переменных, в то время как объекты должны задаваться ссылками. С другой стороны, создание экземпляра с помощью конструктора, вызов методов, наконец, существование родительского класса Object, - все это указывает на то, что int - это настоящий класс. Правильный ответ состоит в том, что int - это и тип, и класс. В зависимости от контекста x может восприниматься как переменная типа int или как объект класса int. Это же верно и для всех остальных value-типов. Замечу еще, что все значимые типы фактически реализованы как структуры, представляющие частный случай класса.

Остается понять, для чего в языке C# введена такая двойственность. Для int и других значимых типов сохранена концепция типа не только из-за ностальгических воспоминаний о типах. Дело в том, что значимые типы эффективнее в реализации, им проще отводить память, так что именно соображения эффективности реализации заставили авторов языка сохранить значимые типы. Более важно, что зачастую необходимо оперировать значениями, а не ссылками на них, хотя бы из-за различий в семантике присваивания для переменных ссылочных и значимых типов.

С другой стороны, в определенном контексте крайне полезно рассматривать переменные типа int как настоящие объекты и обращаться с ними как с объектами. В частности, полезно иметь возможность создавать и работать со списками, чьи элементы являются разнородными объектами, в том числе и принадлежащими к значимым типам.

Дальнейшие примеры работы с типами и проект Types

Обсуждение особенностей тех или иных конструкций языка невозможно без приведения примеров. Для каждой лекции я строю один или несколько проектов, сохраняя по возможности одну и ту же схему и реально выполняя проекты в среде Visual Studio .Net. Для работы с примерами данной лекции построен консольный проект с именем Types, содержащий два класса: Class1 и Testing. Расскажу чуть подробнее о той схеме, по которой выстраиваются проекты. Класс Class1 строится автоматически при начальном создании проекта. Он содержит процедуру Main - точку входа в проект. В процедуре Main создается объект класса Testing и вызываются методы этого класса, тестирующие те или иные ситуации. Для решения специальных задач, помимо всегда создаваемого класса Testing,

создаются один или несколько классов. Добавление нового класса в проект я осуществляю выбором пункта меню **Project/Add Class**. В этом случае автоматически строится заготовка для нового класса, содержащая конструктор без параметров. Дальнейшая работа над классом ведется над этой заготовкой. Создаваемые таким образом классы хранятся в проекте в отдельных файлах. Это особенно удобно, если классы используются в разных проектах. Функционально связанную группу классов удобнее хранить в одном файле, что не возбраняется.

Все проекты в книге являются самодокументируемыми. Классы и их методы сопровождаются тегами <summary>. В результате появляются подсказки при вызове методов и возможность построения XML-отчета, играющего роль спецификации проекта.

Приведу текст класса Class1:

```
using System;
namespace Types
{
	/// <summary>
	/// Проект Types содержит примеры, иллюстрирующие работу
	/// со встроенными скалярными типами языка C#.
	/// Проект содержит классы: Testing, Class1.
	///
	/// </summary>
		class Class1
	{
		/// <summary>
		/// Точка входа проекта.
		/// В ней создается объект класса Testing
		/// и вызываются его методы.
		/// </summary>
		[STAThread]
		static void Main()
		{
			Testing tm = new Testing();
			Console.WriteLine("Testing.Who Test");
			tm.WhoTest();
			Console.WriteLine("Testing.Back Test");
			tm.BackTest();
			Console.WriteLine("Testing.OLoad Test");
			tm.OLoadTest();
			Console.WriteLine("Testing.ToString Test");
			tm.ToStringTest();
			Console.WriteLine("Testing.FromString Test");
			tm.FromStringTest();
			Console.WriteLine("Testing.CheckUncheck Test");
			tm.CheckUncheckTest();
		}
	}
}
```

Класс Class1 содержит точку входа Main и ничего более. В процедуре Main создается объект tm класса Testing, затем поочередно вызываются семь методов этого класса. Каждому вызову предшествует выдача соответствующего сообщения на консоль. Каждый метод - это отдельный пример, подлежащий обсуждению.

Семантика присваивания

Рассмотрим присваивание:

```
x = e
```

Чтобы присваивание было допустимым, типы переменной x и выражения e должны быть согласованными. Пусть сущность x согласно объявлению принадлежит классу T. Будем говорить, что тип T основан на классе T и является базовым типом x, так что базовый тип определяется классом объявления. Пусть теперь в рассматриваемом нами присваивании выражение e связано с объектом типа T1.

Определение: тип T1 согласован по присваиванию с базовым типом T переменной x, если класс T1 является потомком класса T.

Присваивание допустимо, если и только если имеет место согласование типов. Так как все классы в языке C# - встроенные и определенные пользователем - по определению являются потомками класса Object, то отсюда и следует наш частный случай - переменным класса Object можно присваивать выражения любого типа.

Несмотря на то, что обстоятельный разговор о наследовании, родителях и потомках нам еще предстоит, лучше с самого начала понимать отношения между родительским классом и классом-потомком, отношения между объектами этих классов. Класс-потомок при создании **наследует** все свойства и методы родителя. Родительский класс не имеет возможности наследовать свойства и методы, создаваемые его потомками. Наследование - это односторонняя операция от родителя к потомку. Ситуация с присваиванием симметричная. Объекту родительского класса **присваивается** объект класса-потомка. Объекту класса-потомка не может быть присвоен объект родительского класса. Присваивание - это односторонняя операция от потомка к родителю. Одностороннее присваивание реально означает, что ссылочная переменная родительского класса может быть связана с любыми объектами, имеющими тип потомков родительского класса.

Например, пусть задан некоторый класс Parent, а класс Child - его потомок, объявленный следующим образом:

```
class Child:Parent {...}
```

Пусть теперь в некотором классе, являющемся клиентом классов Parent и Child, объявлены переменные этих классов и созданы связанные с ними объекты:

```
Parent p1 = new Parent(), p2 = new Parent();
Child ch1 = new Child(), ch2 = new Child();
```

Тогда допустимы присваивания:

```
p1 = p2; p2= p1; ch1=ch2; ch2 = ch1; p1 = ch1; p2 = ch2;
```

Но недопустимы присваивания:

```
ch1 = p1; ch2 = p1; ch2 = p2; ch1 = p2;
```

Заметьте, ситуация не столь удручающая - сын может вернуть себе переданный родителю объект, задав явное преобразование. Так что следующие присваивания допустимы:

```
p1 = ch1; ... ch1 = (Child)p1;
```

Семантика присваивания справедлива и для другого важного случая - при рассмотрении соответствия между формальными и фактическими аргументами процедур и функций. Если формальный аргумент согласно объявлению имеет тип T, а выражение, задающее фактический аргумент, имеет тип T1, то имеет место согласование типов **формального и**

фактического аргумента, если и только если класс T1 является потомком класса T. Отсюда незамедлительно следует, что если формальный параметр процедуры принадлежит классу Object, то фактический аргумент может быть выражением любого типа.

Преобразование к типу object

Рассмотрим частный случай присваивания x = e; когда x имеет тип object. В этом случае гарантируется полная согласованность по присваиванию - выражение e может иметь любой тип. В результате присваивания значением переменной x становится ссылка на объект, заданный выражением e. Заметьте, текущим типом x становится тип объекта, заданного выражением e. Уже здесь проявляется одно из важных различий между классом и типом. Переменная, лучше сказать сущность x, согласно объявлению принадлежит классу Object, но ее тип - тип того объекта, с которым она связана в текущий момент, - может динамически изменяться.

Примеры преобразований

Перейдем к примерам. Класс Testing, содержащий примеры, представляет собой набор данных разного типа, над которыми выполняются операции, иллюстрирующие преобразования типов. Вот описание класса Testing:

```
using System;
namespace Types
{
	/// <summary>
	/// Класс Testing включает данные разных типов. Каждый его
	/// открытый метод описывает некоторый пример,
	/// демонстрирующий работу с типами.
	/// Открытые методы могут вызывать закрытые методы класса.
	/// </summary>
	public class Testing
	{
		/// <summary>
		/// набор скалярных данных разного типа.
		/// </summary>
		byte b = 255;
		int x = 11;
		uint ux = 1111;
		float y = 5.5f;
		double dy = 5.55;
		string s = "Hello!";
		string s1 = "25";
		object obj = new Object();
		// Далее идут методы класса, приводимые по ходу
		// описания примеров
	}
}
```

В набор данных класса входят скалярные данные арифметического типа, относящиеся к значимым типам, переменные строкового типа и типа object, принадлежащие ссылочным типам. Рассмотрим закрытый (private) метод этого класса - процедуру WhoIsWho с формальным аргументом класса Object. Процедура выводит на консоль переданное ей имя аргумента, его тип и значение. Вот ее текст:

```
/// <summary>
/// Метод выводит на консоль информацию о типе и
/// значении фактического аргумента. Формальный
/// аргумент имеет тип object. Фактический аргумент
/// может иметь любой тип, поскольку всегда
/// допустимо неявное преобразование в тип object.
```

```
/// </summary>
/// <param name="name"> - Имя второго аргумента</param>
/// <param name="any"> - Допустим аргумент любого типа</param>
void WhoIsWho(string name, object any)
{
        Console.WriteLine("type {0} is {1} , value is {2}",
                name, any.GetType(), any.ToString());
}
```

Вот открытый (public) метод класса Testing, в котором многократно вызывается метод WhoIsWho с аргументами разного типа:

```
/// <summary>
/// получаем информацию о типе и значении
/// переданного аргумента - переменной или выражения
/// </summary>
public void WhoTest()
{
        WhoIsWho("x",x);
        WhoIsWho("ux",ux);
        WhoIsWho("y",y);
        WhoIsWho("dy",dy);
        WhoIsWho("s",s);
        WhoIsWho("11 + 5.55 + 5.5f",11 + 5.55 + 5.5f);
        obj = 11 + 5.55 + 5.5f;
        WhoIsWho("obj",obj);
}
```

Заметьте, сущность any - формальный аргумент класса Object при каждом вызове - динамически изменяет тип, связываясь с объектом, заданным фактическим аргументом. Поэтому тип аргумента, выдаваемый на консоль, - это тип фактического аргумента. Заметьте также, что наследуемый от класса Object метод GetType возвращает тип FCL, то есть тот тип, на который отражается тип языка и с которым реально идет работа при выполнении модуля. В большинстве вызовов фактическим аргументом является переменная - соответствующее свойство класса Testing, но в одном случае передается обычное арифметическое выражение, автоматически преобразуемое в объект. Аналогичная ситуация имеет место и при выполнении присваивания в рассматриваемой процедуре.

На рис. 3.1 показаны результаты вывода на консоль, полученные при вызове метода WhoTest в приведенной выше процедуре Main класса Class1.

```
Testing.Who Test
type x is System.Int32 , value is 11
type ux is System.UInt32 , value is 1111
type y is System.Single , value is 5.5
type dy is System.Double , value is 5.55
type s is System.String , value is Hello!
type 11 + 5.55 + 5.5f is System.Double , value is 22.05
type obj is System.Double , value is 22.05
```

Рис. 3.1. Вывод на печать результатов теста WhoTest

Семантика присваивания. Преобразования между ссылочными и значимыми типами

Рассматривая семантику присваивания и передачи аргументов, мы обошли молчанием один важный вопрос. Будем называть целью левую часть оператора присваивания, а также формальный аргумент при передаче аргументов в процедуру или функцию. Будем называть источником правую часть оператора присваивания, а также фактический аргумент при передаче аргументов в процедуру или функцию. Поскольку источник и цель могут быть как значимого, так и ссылочного типа, то возможны четыре различные комбинации.

Рассмотрим их подробнее.

- Цель **и** источник значимого типа. Здесь наличествует семантика значимого присваивания. В этом случае источник и цель имеют собственную память для хранения значений. Значения источника заменяют значения соответствующих полей цели. Источник и цель после этого продолжают жить независимо. У них своя память, хранящая после присваивания одинаковые значения.
- Цель **и** источник ссылочного типа. Здесь имеет место семантика ссылочного присваивания. В этом случае значениями источника и цели являются ссылки на объекты, хранящиеся в памяти ("куче"). При ссылочном присваивании цель разрывает связь с тем объектом, на который она ссылалась до присваивания, и становится ссылкой на объект, связанный с источником. Результат ссылочного присваивания двоякий. Объект, на который ссылалась цель, теряет одну из своих ссылок и может стать висячим, так что его дальнейшую судьбу определит сборщик мусора. С объектом в памяти, на который ссылался источник, теперь связываются, по меньшей мере, две ссылки, рассматриваемые как различные имена одного объекта. Ссылочное присваивание приводит к созданию псевдонимов - к появлению разных имен у одного объекта. Особо следует учитывать ситуацию, когда цель и/или источник имеет значение void. Если такое значение имеет источник, то в результате присваивания цель получает это значение и более не ссылается ни на какой объект. Если же цель имела значение void, а источник - нет, то в результате присваивания ранее "висячая" цель становится ссылкой на объект, связанный с источником.
- Цель ссылочного типа, источник значимого типа. В этом случае "на лету" значимый тип преобразуется в ссылочный. Как обеспечивается двойственность существования значимого и ссылочного типа - переменной и объекта? Ответ прост: за счет специальных, эффективно реализованных операций, преобразующих переменную значимого типа в объект и обратно. Операция " упаковать " (boxing) выполняется автоматически и неявно в тот момент, когда по контексту требуется объект, а не переменная. Например, при вызове процедуры WhoIsWho требуется, чтобы аргумент any был объектом. Если фактический аргумент является переменной значимого типа, то автоматически выполняется операция " упаковать ". При ее выполнении создается настоящий объект, хранящий значение переменной. Можно считать, что происходит упаковка переменной в объект. Необходимость в упаковке возникает достаточно часто. Примером может служить и процедура консольного вывода WriteLine класса Console, которой требуются объекты, а передаются зачастую переменные значимого типа.
- Цель значимого типа, источник ссылочного типа. В этом случае "на лету" ссылочный тип преобразуется в значимый. Операция " распаковать " (unboxing) выполняет обратную операцию, - она "сдирает" объектную упаковку и извлекает хранимое значение. Заметьте, операция " распаковать " не является обратной к операции " упаковать " в строгом смысле этого слова. Оператор obj = x корректен, но выполняемый следом оператор x = obj приведет к ошибке. Недостаточно, чтобы хранимое значение в упакованном объекте точно совпадало по типу с переменной, которой присваивается объект. Необходимо явно заданное преобразование к нужному типу.

Операции "упаковать" и "распаковать" (boxing и unboxing).

Примеры

В нашем следующем примере демонстрируется применение обеих операций - упаковки и распаковки. Поскольку формальный аргумент процедуры Back принадлежит классу Object, то при передаче фактического аргумента значимого типа происходит упаковка значения в объект. Этот объект и возвращается процедурой. Его динамический тип определяется тем

объектом памяти, на который указывает ссылка. Когда возвращаемый результат присваивается переменной значимого типа, то, несмотря на совпадение типа переменной с динамическим типом объекта, необходимо выполнить распаковку, "содрать" объектную упаковку и вернуть непосредственное значение. Вот как выглядит процедура Back и тестирующая ее процедура BackTest из класса Testing:

```
/// <summary>
/// Возвращает переданный ему аргумент.
/// Фактический аргумент может иметь произвольный тип.
/// Возвращается всегда объект класса object.
/// Клиент, вызывающий метод, должен при необходимости
/// задать явное преобразование получаемого результата
/// </summary>
/// <param name="any"> Допустим любой аргумент</param>
/// <returns></returns>
object Back(object any)
{
        return(any);
}
/// <summary>
/// Неявное преобразование аргумента в тип object
/// Явное приведение типа результата.
/// </summary>
public void BackTest()
{
        ux = (uint)Back(ux);
        WhoIsWho("ux",ux);
        s1 = (string)Back(s);
        WhoIsWho("s1",s1);
        x =(int)(uint)Back(ux);
        WhoIsWho("x",x);
        y = (float)(double)Back(11 + 5.55 + 5.5f);
        WhoIsWho("y",y);
}
```

Обратите внимание, что если значимый тип в левой части оператора присваивания не совпадает с динамическим типом объекта, то могут потребоваться две операции приведения. Вначале нужно распаковать значение, а затем привести его к нужному типу, что и происходит в двух последних операторах присваивания. Приведу результаты вывода на консоль, полученные при вызове процедуры BackTest в процедуре Main.

```
Testing.Back Test
type ux is System.UInt32 , value is 1111
type s1 is System.String , value is Hello!
type x is System.Int32 , value is 1111
type y is System.Single , value is 22,05
```

Рис. 3.2. Вывод на печать результатов теста BackTest

Две двойственные операции " упаковать " и " распаковать " позволяют, в зависимости от контекста, рассматривать значимые типы как ссылочные, переменные как объекты, и наоборот.

4. Лекция: Преобразования типов: версия для печати и PDA

Преобразования типов. Преобразования внутри арифметического типа. Преобразования строкового типа. Класс Convert и его методы. Проверяемые преобразования. Управление проверкой арифметических преобразований.

Где, как и когда выполняются преобразования типов?

Необходимость в преобразовании типов возникает в **выражениях**, **присваиваниях**, **замене формальных аргументов** метода **фактическими**.

Если при вычислении выражения операнды операции имеют разные типы, то возникает необходимость приведения их к одному типу. Такая необходимость возникает и тогда, когда операнды имеют один тип, но он несогласован с типом операции. Например, при выполнении сложения операнды типа byte должны быть приведены к типу int, поскольку сложение не определено над байтами. При выполнении присваивания x=e тип источника e и тип цели x должны быть согласованы. Аналогично, при вызове метода также должны быть согласованы типы источника и цели - фактического и формального аргументов.

Преобразования ссылочных типов

Поскольку операции над ссылочными типами не определены (исключением являются строки, но операции над ними, в том числе и присваивание, выполняются как над значимыми типами), то необходимость в них возникает только при присваиваниях и вызовах методов. Семантика таких преобразований рассмотрена в предыдущей лекции 3, где подробно обсуждалась семантика присваивания и совпадающая с ней семантика замены формальных аргументов фактическими. Там же много говорилось о преобразованиях между ссылочными и значимыми типами, выполняемых при этом операциях упаковки значений в объекты и обратной их распаковки.

Коротко повторю основные положения, связанные с преобразованиями ссылочных типов. При присваиваниях (замене аргументов) тип источника должен быть согласован с типом цели, то есть объект, связанный с источником, должен принадлежать классу, являющемуся потомком класса цели. В случае согласования типов, ссылочная переменная цели связывается с объектом источника и ее тип динамически изменяется, становясь типом источника. Это преобразование выполняется автоматически и неявно, не требуя от программиста никаких дополнительных указаний. Если же тип цели является потомком типа источника, то неявное преобразование отсутствует, даже если объект, связанный с источником, принадлежит типу цели. Явное преобразование, заданное программистом, позволяет справиться с этим случаем. Ответственность за корректность явных преобразований лежит на программисте, так что может возникнуть ошибка на этапе выполнения, если связываемый объект реально не является объектом класса цели. За примерами следует обратиться к лекции 3, еще раз обратив внимание на присваивания объектов классов Parent и Child.

Преобразования типов в выражениях

Продолжая тему преобразований типов, рассмотрим привычные для программистов преобразования между значимыми типами и, прежде всего, преобразования внутри арифметического типа.

В C# такие преобразования делятся на неявные и явные. К неявным относятся те преобразования, результат выполнения которых всегда успешен и не приводит к потере точности данных. Неявные преобразования выполняются автоматически. Для арифметических данных это означает, что в неявных преобразованиях диапазон типа назначения содержит в себе диапазон исходного типа. Например, преобразование из типа byte в тип int относится к неявным, поскольку диапазон типа byte является подмножеством диапазона int. Это преобразование всегда успешно и не может приводить к потере точности. Заметьте, преобразования из целочисленных типов к типам с плавающей точкой относятся к неявным. Хотя здесь и может происходить некоторое искажение значения, но точность

представления значения сохраняется, например, при преобразовании из long в double порядок значения остается неизменным.

К явным относятся разрешенные преобразования, успех выполнения которых не гарантируется или может приводить к потере точности. Такие потенциально опасные преобразования должны быть явно заданы программистом. Преобразование из типа int в тип byte относится к явным, поскольку оно небезопасно и может приводить к потере значащих цифр. Заметьте, не для всех типов существуют явные преобразования. В этом случае требуются другие механизмы преобразования типов, которые будут рассмотрены позже.

Преобразования внутри арифметического типа

Арифметический тип, как показано в таблице 3.1, распадается на 11 подтипов. На рис. 4.1 показана схема преобразований внутри арифметического типа.

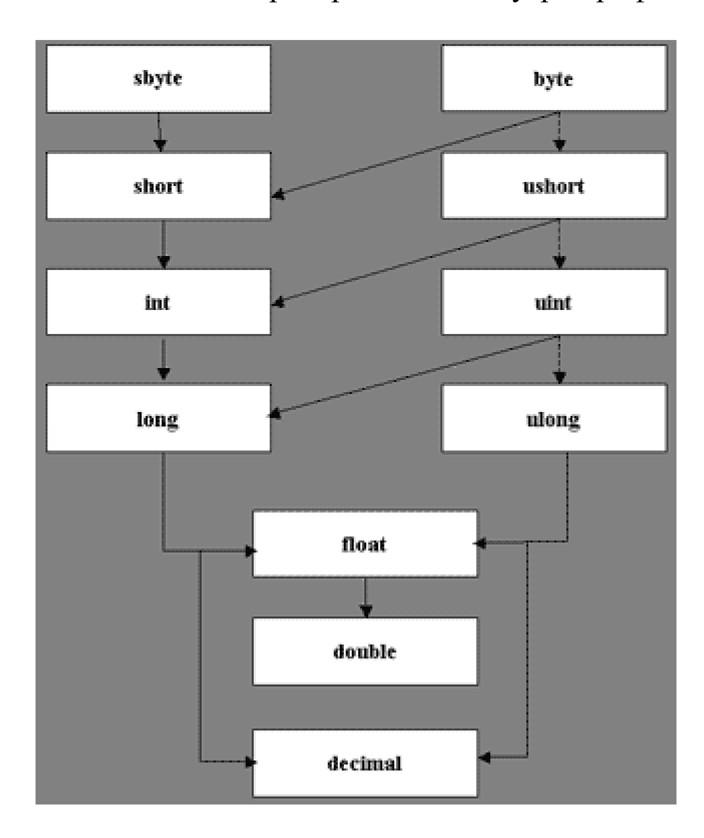


Рис. 4.1. Иерархия преобразований внутри арифметического типа

Диаграмма, приведенная на рисунке, позволяет ответить на ряд важных вопросов, связанных с существованием преобразований между типами. Если на диаграмме задан путь (стрелками) от типа А к типу В, то это означает существование неявного преобразования из типа А в тип В. Все остальные преобразования между подтипами арифметического типа существуют, но являются явными. Заметьте, что циклов на диаграмме нет, все стрелки односторонние, так что преобразование, обратное к неявному, всегда должно быть задано явным образом.

Путь, указанный на диаграмме, может быть достаточно длинным, но это вовсе не означает, что выполняется вся последовательность преобразований на данном пути. Наличие пути говорит лишь о существовании неявного преобразования, а само преобразование выполняется только один раз, - из типа источника А в тип назначения В.

Иногда возникает ситуация, при которой для одного типа источника может одновременно существовать несколько типов назначений и необходимо осуществить выбор цели - типа назначения. Такие проблемы выбора возникают, например, при работе с перегруженными методами в классах.

Понятие перегрузки методов и операций подробно будет рассмотрено в последующих лекциях (см. лекцию 9).

Диаграмма, приведенная на рис. 4.1, и в этом случае помогает понять, как делается выбор. Пусть существует две или более реализации перегруженного метода, отличающиеся типом формального аргумента. Тогда при вызове этого метода с аргументом типа T может возникнуть проблема, какую реализацию выбрать, поскольку для нескольких реализаций может быть допустимым преобразование аргумента типа T в тип, заданный формальным аргументом данной реализации метода. Правило выбора реализации при вызове метода таково: выбирается та реализация, для которой путь преобразований, заданный на диаграмме, короче. Если есть точное соответствие параметров по типу (путь длины 0), то, естественно, именно эта реализация и будет выбрана.

Давайте рассмотрим еще один тестовый пример. В класс Testing включена группа перегруженных методов OLoad с одним и двумя аргументами. Вот эти методы:

```
/// <summary>
/// Группа перегруженных методов OLoad
/// с одним или двумя аргументами арифметического типа.
/// Если фактический аргумент один, то будет вызван один из
/// методов, наиболее близко подходящий по типу аргумента.
/// При вызове метода с двумя аргументами возможен
/// конфликт выбора подходящего метода, приводящий
/// к ошибке периода компиляции.
/// </summary>
void OLoad(float par)
{
        Console.WriteLine("float value {0}", par);
}
/// <summary>
/// Перегруженный метод OLoad с одним параметром типа long
/// </summary>
/// <param name="par"></param>
void OLoad(long par)
{
        Console.WriteLine("long value {0}", par);
}
/// <summary>
/// Перегруженный метод OLoad с одним параметром типа ulong
/// </summary>
/// <param name="par"></param>
void OLoad(ulong par)
{
        Console.WriteLine("ulong value {0}", par);
}
/// <summary>
/// Перегруженный метод OLoad с одним параметром типа double
/// </summary>
/// <param name="par"></param>
void OLoad(double par)
{
        Console.WriteLine("double value {0}", par);
}
/// <summary>
/// Перегруженный метод OLoad с двумя параметрами типа long и long
/// </summary>
```

```
/// <param name="par1"></param>
/// <param name="par2"></param>
void OLoad(long par1, long par2)
{
        Console.WriteLine("long par1 {0}, long par2 {1}", par1, par2);
}
/// <summary>
/// Перегруженный метод OLoad с двумя параметрами типа
/// double и double
/// </summary>
/// <param name="par1"></param>
/// <param name="par2"></param>
void OLoad(double par1, double par2)
{
        Console.WriteLine("double par1 {0}, double par2 {1}",par1, par2);
}
/// <summary>
/// Перегруженный метод OLoad с двумя параметрами типа
/// int и float
/// </summary>
/// <param name="par1"></param>
/// <param name="par2"></param>
void OLoad(int par1, float par2)
{
        Console.WriteLine("int par1 {0}, float par2 {1}",par1, par2);
}
```

Все эти методы устроены достаточно просто. Они сообщают информацию о типе и значении переданных аргументов. Вот тестирующая процедура, вызывающая метод OLoad с разным числом и типами аргументов:

```
/// <summary>
/// Вызов перегруженного метода OLoad. В зависимости от
/// типа и числа аргументов вызывается один из методов группы.
/// </summary>
public void OLoadTest()
{
        OLoad(x); OLoad(ux);
        OLoad(y); OLoad(dy);
        // OLoad(x,ux);
        // conflict: (int, float) и (long,long)
        OLoad(x,(float)ux);
        OLoad(y,dy); OLoad(x,dy);
}
```

Заметьте, один из вызовов закомментирован, так как он приводит к конфликту на этапе трансляции. Для устранения конфликта при вызове метода пришлось задать явное преобразование аргумента, что показано в строке, следующей за строкой-комментарием.

```
Testing.OLoad Test
long value 1111
long value 1111
float value 22,05
double value 5,55
int par1 1111, float par2 1111
double par1 22,0499992370605, double par2 5,55
double par1 1111, double par2 5,55
```

Рис. 4.2. Вывод на печать результатов теста OLoadTest

Прежде чем посмотреть на результаты работы тестирующей процедуры, попробуйте понять, какой из перегруженных методов вызывается для каждого из вызовов. В случае

каких-либо сомнений используйте схему, приведенную на 4.1.

Приведу все-таки некоторые комментарии. При первом вызове метода тип источника - int, а тип аргумента у четырех возможных реализаций соответственно float, long, ulong, double. Явного соответствия нет, поэтому нужно искать самый короткий путь на схеме. Так как не существует неявного преобразования из типа int в тип ulong (на диаграмме нет пути), то остаются возможными три реализации. Но путь из int в long короче, чем остальные пути, поэтому будет выбрана long-реализация метода.

Следующий вызов демонстрирует еще одну возможную ситуацию. Для типа источника uint существуют две возможные реализации, и пути преобразований для них имеют одинаковую длину. В этом случае выбирается та реализация, для которой на диаграмме путь показан сплошной, а не пунктирной стрелкой, потому будет выбрана реализация с параметром long.

Рассмотрим еще ситуацию, приводящую к конфликту. Первый аргумент в соответствии с правилами требует вызова одной реализации, а второй аргумент будет настаивать на вызове другой реализации. Возникнет коллизия, не разрешимая правилами C# и приводящая к ошибке периода компиляции. Коллизию требуется устранить, например, как это сделано в примере. Обратите внимание - обе реализации допустимы, и существуй даже только одна из них, ошибки бы не возникало.

Явные преобразования

Как уже говорилось, явные преобразования могут быть опасными из-за потери точности. Поэтому они выполняются по указанию программиста, - на нем лежит вся ответственность за результаты.

Преобразования строкового типа

Важным классом преобразований являются **преобразования в строковый тип** и наоборот. Преобразования в строковый тип всегда определены, поскольку, напомню, все типы являются потомками базового класса Object, а, следовательно, обладают методом ToString(). Для встроенных типов определена подходящая реализация этого метода. В частности, для всех подтипов арифметического типа метод ToString() возвращает в подходящей форме строку, задающую соответствующее значение арифметического типа. Заметьте, метод ToString можно вызывать явно, но, если явный вызов не указан, то он будет вызываться неявно, всякий раз, когда по контексту требуется преобразование к строковому типу. Вот соответствующий пример:

```
/// <summary>
/// Демонстрация преобразования в строку данных различного типа.
/// </summary>
public void ToStringTest()
{
        s ="Владимир Петров ";
        s1 =" Возраст: "; ux = 27;
        s = s + s1 + ux.ToString();
        s1 =" Зарплата: "; dy = 2700.50;
        s = s + s1 + dy;
        WhoIsWho("s",s);
}
```

```
Testing.ToString Test
type s is System.String , value is Владимир Петров  Возраст: 27 Зарплата: 2700,5
```

Рис. 4.3. Вывод на печать результатов теста ToStringTest

Здесь для переменной ux метод был вызван явно, а для переменной dy он вызывается автоматически. Результат работы этой процедуры показан на рис. 4.3.

Преобразования из строкового типа в другие типы, например, в арифметический, должны выполняться явно. Но явных преобразований между арифметикой и строками не существует. Необходимы другие механизмы, и они в C# имеются. Для этой цели можно использовать соответствующие методы класса Convert библиотеки FCL, встроенного в пространство имен System. Приведу соответствующий пример:

```
/// <summary>
/// Демонстрация преобразования строки в данные различного типа.
/// </summary>
public void FromStringTest()
{
        s ="Введите возраст ";
        Console.WriteLine(s);
        s1 = Console.ReadLine();
        ux = Convert.ToUInt32(s1);
        WhoIsWho("Возраст: ",ux);
        s ="Введите зарплату ";
        Console.WriteLine(s);
        s1 = Console.ReadLine();
        dy = Convert.ToDouble(s1);
        WhoIsWho("Зарплата: ",dy);
}
```

Этот пример демонстрирует ввод с консоли данных разных типов. Данные, читаемые с консоли методом ReadLine или Read, всегда представляют собой строку, которую затем необходимо преобразовать в нужный тип. Тут-то и вызываются соответствующие методы класса Convert. Естественно, для успеха преобразования строка должна содержать значение в формате, допускающем подобное преобразование. Заметьте, например, что при записи значения числа для выделения дробной части должна использоваться запятая, а не точка; в противном случае возникнет ошибка периода выполнения.

В различных версиях Visual Studio возможны разные разделители целой и дробной частей, они также зависят от региональных настроек в ОС.

На рис. 4.4 показаны результаты вывода и ввода данных с консоли при работе этой процедуры.

```
Testing.FromString Test
Введите возраст
27
type Возраст:  is System.UInt32 , value is 27
Введите зарплату
7800
type Зарплата:  is System.Double , value is 7800
```

Рис. 4.4. Вывод на печать результатов теста FromStringTest

Преобразования и класс Convert

Класс Convert, определенный в пространстве имен System, играет важную роль, обеспечивая необходимые преобразования между различными типами. Напомню, что внутри арифметического типа можно использовать более простой, скобочный способ приведения к нужному типу. Но таким способом нельзя привести, например, переменную типа string к типу int, оператор присваивания: ux = (int)s1; приведет к ошибке периода компиляции. Здесь необходим вызов метода ToInt32 класса Convert, как это сделано в

последнем примере предыдущего раздела.

Методы класса Convert поддерживают общий способ выполнения преобразований между типами. Класс Convert содержит 15 статических методов вида To <Type> (ToBoolean(),...ToUInt64()), где Type может принимать значения от Boolean до UInt64 для всех встроенных типов, перечисленных в таблице 3.1. Единственным исключением является тип object, - метода ToObject нет по понятным причинам, поскольку для всех типов существует неявное преобразование к типу object. Среди других методов отмечу общий статический метод ChangeType, позволяющий преобразование объекта к некоторому заданному типу.

Существует возможность преобразования к системному типу DateTime, который хотя и не является встроенным типом языка C#, но допустим в программах, как и любой другой системный тип. Приведу простейший пример работы с этим типом:

```
// System type: DateTime
System.DateTime dat = Convert.ToDateTime("15.03.2003");
Console.WriteLine("Date = {0}", dat);
```

Результатом вывода будет строка:

```
Date = 15.03.2003      0:00:00
```

Все методы To <Type> класса Convert перегружены и каждый из них имеет, как правило, более десятка реализаций с аргументами разного типа. Так что фактически эти методы задают все возможные преобразования между всеми встроенными типами языка C#.

Кроме методов, задающих преобразования типов, в классе Convert имеются и другие методы, например, задающие преобразования символов Unicode в однобайтную кодировку ASCII, преобразования значений объектов и другие методы. Подробности можно посмотреть в справочной системе.

Проверяемые преобразования

Уже упоминалось о том, что при выполнении явных преобразований могут возникать нежелательные явления, например, потеря точности. Я говорил, что вся ответственность за это ложится на программиста, и легче ему от этого не становится. А какую часть этого бремени может взять на себя язык программирования? Что можно предусмотреть для обнаружения ситуаций, когда такие явления все-таки возникают? В языке C# имеются необходимые для этого средства.

Язык C# позволяет создать проверяемый блок, в котором будет осуществляться проверка результата вычисления арифметических выражений. Если результат вычисления значения источника выходит за диапазон возможных значений целевой переменной, то возникнет исключение (говорят также: "будет выброшено исключение ") соответствующего типа. Если предусмотрена обработка исключения, то дальнейшее зависит от обработчика исключения. В лучшем случае, программа сможет продолжить корректное выполнение. В худшем, - она остановится и выдаст информацию об ошибке. Заметьте, не произойдет самого опасного - продолжения работы программы с неверными данными.

Синтаксически проверяемый блок предваряется ключевым словом checked. В теле такого блока арифметические преобразования проверяются на допустимость. Естественно, подобная проверка требует дополнительных временных затрат. Если группа операторов в теле такого блока нам кажется безопасной, то их можно выделить в непроверяемый блок, используя ключевое слово unchecked. Замечу еще, что и в непроверяемом блоке при работе

методов Convert все опасные преобразования проверяются и приводят к выбрасыванию исключений. Приведу пример, демонстрирующий все описанные ситуации:

```
/// <summary>
/// Демонстрация проверяемых и непроверяемых преобразований.
/// Опасные проверяемые преобразования приводят к исключениям.
/// Опасные непроверяемые преобразования приводят к неверным
/// результатам, что совсем плохо.
/// </summary>
public void CheckUncheckTest()
{
        x = -25^2;
        WhoIsWho ("x", x);
        b= 255;
        WhoIsWho("b",b);
        // Проверяемые опасные преобразования.
        // Возникают исключения, перехватываемые catch-блоком.
        checked
        {
                try
                {
                        b += 1;
                }
                catch (Exception e)
                {
                        Console.WriteLine("Переполнение при вычислении b");
                        Console.WriteLine(e);
                }
                try
                {
                        b = (byte)x;
                }
                catch (Exception e)
                {
                        Console.WriteLine("Переполнение при преобразовании к
byte");
                        Console.WriteLine(e);
                }
                // непроверяемые опасные преобразования
                unchecked
                {
                        try
                        {
                                b +=1;
                                WhoIsWho ("b", b);
                                b = (byte)x;
                                WhoIsWho ("b", b);
                                ux= (uint)x;
                                WhoIsWho ("ux", ux);
                                Console.WriteLine("Исключений нет, но результаты
не верны!");
                        }
                        catch (Exception e)
                        {
                                Console.WriteLine("Этот текст не должен
появляться");
                                Console.WriteLine(e);
                        }
                        // автоматическая проверка преобразований в Convert
                        // исключения возникают, несмотря на unchecked
                        try
                        {
                                b = Convert.ToByte(x);
                        }
                        catch (Exception e)
                        {
```

```
                        Console.WriteLine("Переполнение при
                                преобразовании к byte!");
                        Console.WriteLine(e);
                }
                try
                {
                        ux= Convert.ToUInt32(x);
                }
                catch (Exception e)
                {
                        Console.WriteLine("Потеря знака при
                                преобразовании к uint!");
                        Console.WriteLine(e);
                }
        }
    }
}
```

Исключения и охраняемые блоки. Первое знакомство

В этом примере мы впервые встречаемся с охраняемыми try-блоками. Исключениям и способам их обработки посвящена отдельная лекция, но не стоит откладывать надолго знакомство со столь важным механизмом. Как показывает практика программирования, любая вызываемая программа не гарантирует, что в процессе ее работы не возникнут какие-либо неполадки, в результате которых она не сможет выполнить свою часть контракта. Исключения являются нормальным способом уведомления об ошибках в работе программы. Возникновение ошибки в работе программы должно приводить к выбрасыванию исключения соответствующего типа, следствием чего является прерывание нормального хода выполнения программы и передача управления обработчику исключения - стандартному или предусмотренному самой программой.

Заметьте, рекомендуемый стиль программирования в C# отличается от стиля, принятого в языках C/C++, где функция, в которой возникла ошибка, завершается нормальным образом, уведомляя об ошибке в возвращаемом значении результата. Вызывающая программа должна анализировать результат, чтобы понять, была ли ошибка в работе вызванной функции и какова ее природа. При программировании в стиле C# ответственность за обнаружение ошибок лежит на вызванной программе. Она должна не только обнаружить ошибку, но и явно сообщить о ней, выбрасывая исключение соответствующего типа. Вызываемая программа должна попытаться исправить последствия ошибки в обработчике исключения. Подробности смотрите в лекции про исключения.

В состав библиотеки FCL входит класс Exception, свойства и методы которого позволяют работать с исключениями как с объектами, получать нужную информацию, дополнять объект собственной информацией. У класса Exception - большое число потомков, каждый из которых описывает определенный тип исключения. При проектировании собственных классов можно параллельно проектировать и классы, задающие собственный тип исключений, который может выбрасываться в случае ошибок при работе методов класса. Создаваемый класс исключений должен быть потомком класса Exception.

Если в некотором модуле предполагается возможность появления исключений, то разумно предусмотреть и их обработку. В этом случае в модуле создается охраняемый **try-блок**, предваряемый ключевым словом try. Вслед за этим блоком следуют один или несколько блоков, обрабатывающих исключения, - catch-блоков. Каждый catch-блок имеет формальный параметр класса Exception или одного из его потомков. Если в try-блоке возникает исключение типа T, то catch-блоки начинают конкурировать в борьбе за перехват исключения. Первый по порядку catch-блок, тип формального аргумента которого

согласован с типом T - совпадает с ним или является его потомком - захватывает исключение и начинает выполняться; поэтому порядок написания catch-блоков небезразличен. Вначале должны идти специализированные обработчики. Универсальным обработчиком является catch-блок с формальным параметром родового класса Exception, согласованным с исключением любого типа T. Универсальный обработчик, если он есть, стоит последним, поскольку захватывает исключение любого типа.

Конечно, плохо, когда в процессе работы той или иной процедуры возникает исключение. Однако его появление еще не означает, что процедура не сможет выполнить свой контракт. Исключение может быть нужным образом обработано, после чего продолжится нормальный ход вычислений приложения. Гораздо хуже, когда возникают ошибки в работе процедуры, не приводящие к исключениям. Тогда работа продолжается с неверными данными без исправления ситуации и даже без уведомления о возникновении ошибки. Наш пример показывает, что вычисления в C# могут быть небезопасными и следует применять специальные средства языка, такие как, например, checked-блоки, чтобы избежать появления подобных ситуаций.

Вернемся к обсуждению нашего примера. Здесь как в проверяемых, так и в непроверяемых блоках находятся охраняемые блоки с соответствующими обработчиками исключительных ситуаций. Во всех случаях применяется универсальный обработчик, захватывающий любое исключение в случае его возникновения в try-блоке. Сами обработчики являются простыми уведомителями, они лишь сообщают об ошибочной ситуации, не пытаясь исправить ее.

Опасные вычисления в охраняемых проверяемых блоках

Такая ситуация возникает в первых двух try-блоках нашего примера. Эти блоки встроены в проверяемый checked-блок. В каждом из них используются опасные вычисления, приводящие к неверным результатам. Так, при присваивании невинного выражения b+1 из-за переполнения переменная b получает значение 0, а не 256. Поскольку вычисление находится в проверяемом блоке, то ошибка обнаруживается и результатом является вызов исключения. Далее, поскольку все это происходит в охраняемом блоке, то управление перехватывается и обрабатывается в соответствующем catch-блоке. Эту ситуацию следует отнести к нормальному, разумно построенному процессу вычислений.

Опасные вычисления в охраняемых непроверяемых блоках

Такую ситуацию демонстрирует третий try-блок нашего примера, встроенный в непроверяемый unchecked-блок. Здесь участвуют те же самые опасные вычисления, но теперь их корректность не проверяется, они не вызывают исключений, и как следствие, соответствующий catch-блок не вызывается. Результаты вычислений при этом неверны, но никаких уведомлений об этом нет. Это самая плохая ситуация, которая может случиться при работе наших программ.

Заметьте, проверку переполнения в арифметических вычислениях можно включить не только с помощью создания checked-блоков, но и задав свойство checked проекта (по умолчанию, оно выключено). Как правило, это свойство проекта всегда включается в процессе разработки и отладки. В законченной версии проекта свойство вновь отключается, поскольку полная проверка всех преобразований требует определенных накладных расходов, увеличивая время работы; а проверяемые блоки остаются лишь там, где такой контроль действительно необходим.

Область действия проверки или ее отключения можно распространить и на отдельное выражение. В этом случае спецификаторы checked и unchecked предшествуют выражению, заключенному в круглые скобки. Такое выражение называется проверяемым

(непроверяемым) **выражением**, а checked и unchecked рассматриваются как операции, допустимые в выражениях.

Опасные преобразования и методы класса Convert

Явно выполняемые преобразования по определению относятся к опасным. Явные преобразования можно выполнять по-разному. Синтаксически наиболее просто выполнить приведение типа - кастинг, явно указав тип приведения, как это сделано в только что рассмотренном примере. Но если это делается в непроверяемом блоке, последствия могут быть самыми печальными. Поэтому такой способ приведения типов следует применять с большой осторожностью. Надежнее выполнять преобразования типов более универсальным способом, используя стандартный встроенный класс Convert, специально спроектированный для этих целей.

В нашем примере четвертый и пятый try-блоки встроены в непроверяемый unchecked-блок. Но опасные преобразования реализуются методами класса Convert, которые сами проводят проверку и при необходимости выбрасывают исключения, что и происходит в нашем случае.

На рис. 4.5 показаны результаты работы процедуры CheckUncheckTest. Их анализ способствует лучшему пониманию рассмотренных нами ситуаций.

```
E:\from_D\C#Projects\Types\bin\Debug\Types.exe
Testing.CheckUncheck Test
type x is System.Int32 , value is -27
type b is System.Byte , value is 255
Переполнение при вычислении b!
System.OverflowException: Arithmetic operation resulted in an overflow.
   at Types.Testing.CheckUncheckTest() in e:\from_d\c#projects\types\testing.cs:
line 118
Переполнение при преобразовании к типу byte!
System.OverflowException: Arithmetic operation resulted in an overflow.
   at Types.Testing.CheckUncheckTest() in e:\from_d\c#projects\types\testing.cs:
line 127
type b is System.Byte , value is 0
type b is System.Byte , value is 229
type ux is System.Int32 , value is -27
Исключения не возникают, но результаты не верны!
Переполнение при преобразовании к типу byte!
System.OverflowException: Value was either too large or too small for an unsigne
d byte.
   at System.Convert.ToByte(Int32 value)
   at Types.Testing.CheckUncheckTest() in e:\from_d\c#projects\types\testing.cs:
line 157
Потеря знака при преобразовании к типу uint!
System.OverflowException: Value was either too large or too small for a UInt32.
   at System.Convert.ToUInt32(Int32 value)
   at Types.Testing.CheckUncheckTest() in e:\from_d\c#projects\types\testing.cs:
line 167
```

Рис. 4.5. Вывод на печать результатов теста CheckUncheckTest

На этом, пожалуй, пора поставить точку в обсуждении системы типов языка C#. За получением тех или иных подробностей, как всегда, следует обращаться к справочной системе.

5. Лекция: Переменные и выражения: версия для печати и PDA

Объявление переменных. Синтаксис объявления. Инициализация. Время жизни и область видимости. Где объявляются переменные? Локальные и глобальные переменные. Есть ли глобальные переменные в C#? Константы.

Объявление переменных

В лекции 4 рассматривались типы языка C#. Естественным продолжением этой темы является рассмотрение переменных языка. Переменные и типы - тесно связанные понятия. С объектной точки зрения переменная - это экземпляр типа. Скалярную переменную можно рассматривать как сущность, обладающую именем, значением и типом. Имя и тип задаются при объявлении переменной и остаются неизменными на все время ее жизни. Значение переменной может меняться в ходе вычислений, эта возможность вариации значений и дала имя понятию переменная **(Variable)** в математике и программировании. Получение начального значения переменной называется ее инициализацией. Важной новинкой языка C# является требование обязательной инициализации переменной до начала ее использования. Попытка использовать неинициализированную переменную приводит к ошибкам, обнаруживаемым еще на этапе компиляции. Инициализация переменных, как правило, выполняется в момент объявления, хотя и может быть отложена.

Тесная связь типов и классов в языке C# обсуждалась в предыдущей лекции. Не менее тесная связь существует между переменными и объектами. Так что, когда речь идет о переменной значимого типа, то во многих ситуациях она может играть роль объекта некоторого класса. В этой лекции обсуждение будет связано со скалярными переменными встроенных типов. Все переменные, прежде чем появиться в вычислениях, должны быть объявлены. Давайте рассмотрим, как это делается в C#.

Проект Variables

Как обычно, для рассмотрения примеров построен специальный проект. В данной лекции это консольный проект с именем Variables. Построенный по умолчанию класс Class1 содержит точку входа Main. Добавленный в проект класс Testing содержит набор скалярных переменных и методов, тестирующих разные аспекты работы со скалярными переменными в C#. В процедуре Main создается объект класса Testing и поочередно вызываются его методы, каждый из которых призван проиллюстрировать те или иные моменты работы.

Синтаксис объявления

Общий синтаксис объявления сущностей в C# похож на синтаксис объявления в C++, хотя и имеет ряд отличий. Вот какова общая структура объявления:

```
[<атрибуты>] [<модификаторы>] <тип> <объявители>;
```

Об атрибутах - этой новинке языка C# - уже шла речь, о них будем говорить и в последующих лекциях курса. Модификаторы будут появляться по мере необходимости. При объявлении переменных чаще всего задаются модификаторы доступа - public, private и другие. Если атрибуты и модификаторы могут и не указываться в объявлении, то задание типа необходимо всегда. Ограничимся пока рассмотрением уже изученных встроенных типов. Когда в роли типа выступают имена типов из таблицы 3.1, это означает, что объявляются простые скалярные переменные. Структурные типы - массивы, перечисления, структуры и другие пользовательские типы - будут изучаться в последующих лекциях.

При объявлении простых переменных указывается их тип и список объявителей, где объявитель - это имя или имя с инициализацией. Список объявителей позволяет в одном объявлении задать несколько переменных одного типа. Если объявитель задается именем переменной, то имеет место объявление с отложенной инициализацией. Хороший стиль программирования предполагает задание инициализации переменной в момент ее

объявления. Инициализацию можно осуществлять двояко - обычным присваиванием или в объектной манере. Во втором случае для переменной используется конструкция new и вызывается конструктор по умолчанию. Процедура SimpleVars класса Testing иллюстрирует различные способы объявления переменных и простейшие вычисления над ними:

```
public void SimpleVars()
{
	//Объявления локальных переменных
	int x, s; //без инициализации
	int y =0, u = 77; //обычный способ инициализации
	//допустимая инициализация
	float w1=0f, w2 = 5.5f, w3 =w1+ w2 + 125.25f;
	//допустимая инициализация в объектном стиле
	int z= new int();
	//Недопустимая инициализация.
	//Конструктор с параметрами не определен
	//int v = new int(77);
		x=u+y; //теперь x инициализирована
	if(x> 5) s = 4;
	for (x=1; x<5; x++)s=5;
	//Инициализация в if и for не рассматривается,
	//поэтому s считается неинициализированной переменной
	//Ошибка компиляции:использование неинициализированной переменной
	//Console.WriteLine("s= {0}",s);
}  //SimpleVars
```

В первой строке объявляются переменные x и s с отложенной инициализацией. Заметьте (и это важно!), что всякая попытка использовать еще не инициализированную переменную в правых частях операторов присваивания, в вызовах функций, вообще в вычислениях приводит к ошибке уже на этапе компиляции.

Последующие объявления переменных эквивалентны по сути, но демонстрируют два стиля инициализации - обычный и объектный. Обычная форма инициализации предпочтительнее не только в силу своей естественности, но она и более эффективна, поскольку в этом случае инициализирующее выражение может быть достаточно сложным, с переменными и функциями. На практике объектный стиль для скалярных переменных используется редко. Вместе с тем полезно понимать, что объявление с инициализацией int y =0 можно рассматривать как создание нового объекта (new) и вызова для него конструктора по умолчанию. При инициализации в объектной форме может быть вызван только конструктор по умолчанию, другие конструкторы с параметрами для встроенных типов не определены. В примере закомментировано объявление переменной v с инициализацией в объектном стиле, приводящее к ошибке, где делается попытка дать переменной значение, передавая его конструктору в качестве параметра.

Откладывать инициализацию не стоит, как показывает пример с переменной s, объявленной с отложенной инициализацией. В вычислениях она дважды получает значение: один раз в операторе if, другой - в операторе цикла for. Тем не менее, при компиляции возникнет ошибка, утверждающая, что в процедуре WriteLine делается попытка использовать неинициализированную переменную s. Связано это с тем, что для операторов if и for на этапе компиляции не вычисляются условия, зависящие от переменных. Поэтому компилятор предполагает худшее - условия ложны, инициализация s в этих операторах не происходит. А за инициализацией наш компилятор следит строго, ты так и знай!

Время жизни и область видимости переменных

Давайте рассмотрим такие важные характеристики переменных, как время их жизни и область видимости. Зададимся вопросом, как долго живут объявленные переменные и в какой области программы видимы их имена? Ответ зависит от того, где и как, в каком

контексте объявлены переменные. В языке C# не так уж много возможностей для объявления переменных, пожалуй, меньше, чем в любом другом языке. Открою "страшную" тайну, - здесь вообще нет настоящих глобальных переменных. Их отсутствие не следует считать некоторым недостатком C#, это достоинство языка. Но обо всем по порядку.

Поля

Первая важнейшая роль переменных, - они задают свойства структур, интерфейсов, классов. В языке C# такие переменные называются полями (fields). Структуры, интерфейсы, классы, поля - рассмотрению этих понятий будет посвящена большая часть этого курса, а сейчас сообщу лишь некоторые минимальные сведения, связанные с рассматриваемой темой. Поля объявляются при описании класса (и его частных случаев - интерфейса, структуры). Когда конструктор класса создает очередной объект - экземпляр класса, то он в динамической памяти создает набор полей, определяемых классом, и записывает в них значения, характеризующие свойства данного конкретного экземпляра. Так что каждый объект в памяти можно рассматривать как набор соответствующих полей класса со своими значениями. Время существования и область видимости полей определяются объектом, которому они принадлежат. Объекты в динамической памяти, с которыми не связана ни одна ссылочная переменная, становятся недоступными. Реально они оканчивают свое существование, когда сборщик мусора (garbage collector) выполнит чистку "кучи". Для значимых типов, к которым принадлежат экземпляры структур, жизнь оканчивается при завершении блока, в котором они объявлены.

Есть одно важное исключение. Некоторые поля могут жить дольше. Если при объявлении класса поле объявлено с модификатором static, то такое поле является частью класса и единственным на все его экземпляры. Поэтому static - поля живут так же долго, как и сам класс. Более подробно эти вопросы будут обсуждаться при рассмотрении классов, структур, интерфейсов.

Глобальные переменные уровня модуля. Существуют ли они в C#?

Где еще могут объявляться переменные? Во многих языках программирования переменные могут объявляться на уровне модуля. Такие переменные называются глобальными. Их область действия распространяется, по крайней мере, на весь модуль. Глобальные переменные играют важную роль, поскольку они обеспечивают весьма эффективный способ обмена информацией между различными частями модуля. Обратная сторона эффективности аппарата глобальных переменных, - их опасность. Если какая-либо процедура, в которой доступна глобальная переменная, некорректно изменит ее значение, то ошибка может проявиться в другой процедуре, использующей эту переменную. Найти причину ошибки бывает чрезвычайно трудно. В таких ситуациях приходится проверять работу многих компонентов модуля.

В языке C# роль модуля играют классы, пространства имен, проекты, решения. Поля классов, о которых шла речь выше, могут рассматриваться как глобальные переменные класса. Но здесь у них особая роль. Данные (поля) являются тем центром, вокруг которого вращается мир класса. Заметьте, каждый экземпляр класса - это отдельный мир. Поля экземпляра (открытые и закрытые) - это глобальная информация, которая доступна всем методам класса, играющим второстепенную роль - они обрабатывают данные.

Статические поля класса хранят информацию, общую для всех экземпляров класса. Они представляют определенную опасность, поскольку каждый экземпляр способен менять их значения.

В других видах модуля - пространствах имен, проектах, решениях - нельзя объявлять переменные. В пространствах имен в языке C# разрешено только объявление классов и их частных случаев: структур, интерфейсов, делегатов, перечислений. Поэтому глобальных переменных уровня модуля, в привычном для других языков программирования смысле, в языке C# нет. Классы не могут обмениваться информацией, используя глобальные переменные. Все взаимодействие между ними обеспечивается способами, стандартными для объектного подхода. Между классами могут существовать два типа отношений - клиентские и наследования, а основной способ инициации вычислений - это вызов метода для объекта-цели или вызов обработчика события. Поля класса и аргументы метода позволяют передавать и получать нужную информацию. Устранение глобальных переменных как источника опасных, трудно находимых ошибок существенно повышает надежность создаваемых на языке C# программных продуктов.

Введем в класс Testing нашего примера три закрытых поля и добавим конструктор с параметрами, инициализирующий значения полей при создании экземпляра класса:

```
//fields
                int x,y; //координаты точки
                string name; //имя точки
                //конструктор с параметрами
                public Testing(int x, int y, string name)
                {
                        this.x = x; this.y = y; this.name = name;
                }
```

В процедуре Main первым делом создается экземпляр класса Testing, а затем вызываются методы класса, тестирующие различные ситуации:

```
Testing ts = new Testing(5,10,"Точка1");
                        ts.SimpleVars();
```

Локальные переменные

Перейдем теперь к рассмотрению локальных переменных. Этому важнейшему виду переменных будет посвящена вся оставшаяся часть данной лекции. Во всех языках программирования, в том числе и в C#, основной контекст, в котором появляются переменные, - это процедуры. Переменные, объявленные на уровне процедуры, называются локальными, - они локализованы в процедуре.

В некоторых языках, например в Паскале, локальные переменные должны быть объявлены в вершине процедурного блока. Иногда это правило заменяется менее жестким, но, по сути, аналогичным правилом, - где бы внутри процедурного блока ни была объявлена переменная, она считается объявленной в вершине блока, и ее область видимости распространяется на весь процедурный блок. В C#, также как и в языке C++, принята другая стратегия. Переменную можно объявлять в любой точке процедурного блока. Область ее видимости распространяется от точки объявления до конца процедурного блока.

На самом деле, ситуация с процедурным блоком в C# не так проста. Процедурный блок имеет сложную структуру; в него могут быть вложены другие блоки, связанные с операторами выбора, цикла и так далее. В каждом таком блоке, в свою очередь, допустимы вложения блоков. В каждом внутреннем блоке допустимы объявления переменных. Переменные, объявленные во внутренних блоках, локализованы именно в этих блоках, их область видимости и время жизни определяются этими блоками. Локальные переменные начинают существовать при достижении вычислений в блоке точки объявления и перестают существовать, когда процесс вычисления завершает выполнение операторов блока. Можно полагать, что для каждого такого блока выполняется так называемый пролог

и эпилог. В прологе локальным переменным отводится память, в эпилоге память освобождается. Фактически ситуация сложнее, поскольку выделение памяти, а следовательно, и начало жизни переменной, объявленной в блоке, происходит не в момент входа в блок, а лишь тогда, когда достигается точка объявления локальной переменной.

Давайте обратимся к примеру. В класс Testing добавлен метод с именем ScopeVar, вызываемый в процедуре Main. Вот код этого метода:

```
/// <summary>
/// Анализ области видимости переменных
/// </summary>
/// <param name="x"></param>
public void ScopeVar(int x)
{
        int y =77; string s = name;
        if (s=="Точка1")
        {
                        int u = 5; int v = u+y; x +=1;
                        Console.WriteLine("y= {0}; u={1};
                                v={2}; x={3}", y,u,v,x);
        }
        else
        {
                        int u= 7; int v= u+y;
                        Console.WriteLine("y= {0}; u={1}; v={2}", y,u,v);
        }
        //Console.WriteLine("y= {0}; u={1}; v={2}",y,u,v);
        //Локальные переменные не могут быть статическими.
        //static int Count = 1;
        //Ошибка: использование sum до объявления
        //Console.WriteLine("x= {0}; sum ={1}", x,sum);
        int i;long sum =0;
        for(i=1; i<x; i++)
        {
                //ошибка: коллизия имен: y
                //float y = 7.7f;
                sum +=i;
        }
        Console.WriteLine("x= {0}; sum ={1}", x,sum);
}//ScopeVar
```

Заметьте, в теле метода встречаются имена полей, аргументов и локальных переменных. Эти имена могут совпадать. Например, имя x имеет поле класса и формальный аргумент метода. Это допустимая ситуация. В языке C# разрешено иметь локальные переменные с именами, совпадающими с именами полей класса, - в нашем примере таким является имя y ; однако, запрещено иметь локальные переменные, имена которых совпадают с именами формальных аргументов. Этот запрет распространяется не только на внешний уровень процедурного блока, что вполне естественно, но и на все внутренние блоки.

В процедурный блок вложены два блока, порожденные оператором if. В каждом из них объявлены переменные с одинаковыми именами u и v. Это корректные объявления, поскольку время существования и области видимости этих переменных не пересекаются. Итак, для невложенных блоков разрешено объявление локальных переменных с одинаковыми именами. Заметьте также, что переменные u и v перестают существовать после выхода из блока, так что операторы печати, расположенные внутри блока, работают корректно, а оператор печати вне блока приводит к ошибке, - u и v здесь не видимы, кончилось время их жизни. По этой причине оператор закомментирован.

Выражение, проверяемое в операторе if, зависит от значения поля name. Значение поля глобально для метода и доступно всюду, если только не перекрывается именем аргумента

(как в случае с полем x) или локальной переменной (как в случае с полем y).

Во многих языках программирования разрешено иметь локальные статические переменные, у которых область видимости определяется блоком, но время их жизни совпадает со временем жизни проекта. При каждом повторном входе в блок такие переменные восстанавливают значение, полученное при предыдущем выходе из блока. В языке C# статическими могут быть только поля, но не локальные переменные. Незаконная попытка объявления static переменной в процедуре ScopeVar закомментирована. Попытка использовать имя переменной в точке, предшествующей ее объявлению, также незаконна и закомментирована.

Глобальные переменные уровня процедуры. Существуют ли?

Поскольку процедурный блок имеет сложную структуру с вложенными внутренними блоками, то и здесь возникает тема глобальных переменных. Переменная, объявленная во внешнем блоке, рассматривается как глобальная по отношению к внутренним блокам. Во всех известных мне языках программирования во внутренних блоках разрешается объявлять переменные с именем, совпадающим с именем глобальной переменной. Конфликт имен снимается за счет того, что локальное внутреннее определение сильнее внешнего. Поэтому область видимости внешней глобальной переменной сужается и не распространяется на те внутренние блоки, где объявлена переменная с подобным именем. Внутри блока действует локальное объявление этого блока, при выходе восстанавливается область действия внешнего имени. В языке C# этот гордиев узел конфликтующих имен разрублен, - во внутренних блоках запрещено использование имен, совпадающих с именем, использованным во внешнем блоке. В нашем примере незаконная попытка объявить во внутреннем блоке уже объявленное имя y закомментирована.

Обратите внимание, что подобные решения, принятые создателями языка C#, не только упрощают жизнь разработчикам транслятора. Они способствуют повышению эффективности программ, а самое главное, повышают надежность программирования на C#.

Отвечая на вопрос, вынесенный в заголовок, следует сказать, что глобальные переменные на уровне процедуры в языке C#, конечно же, есть, но нет конфликта имен между глобальными и локальными переменными на этом уровне. Область видимости глобальных переменных процедурного блока распространяется на весь блок, в котором они объявлены, начиная от точки объявления, и не зависит от существования внутренних блоков. Когда говорят, что в C# нет глобальных переменных, то, прежде всего, имеют в виду их отсутствие на уровне модуля. Уже во вторую очередь речь идет об отсутствии конфликтов имен на процедурном уровне.

Константы

Константы C# могут появляться, как обычно, в виде литералов и именованных констант. Вот пример константы, заданной литералом и стоящей в правой части оператора присваивания:

```
const float y = 7.7f;
```

Значение константы " 7.7f " является одновременно ее именем, оно же позволяет однозначно определить тип константы. Заметьте, иногда, как в данном случае, приходится добавлять к значению специальные символы для точного указания типа. Я не буду останавливаться на этих подробностях. Если возникает необходимость уточнить, как записываются литералы, то достаточно получить справку по этой теме. Делается все так же,

как и в языке С++.

Всюду, где можно объявить переменную, можно объявить и именованную константу. Синтаксис объявления схож. В объявление добавляется модификатор const, инициализация констант обязательна и не может быть отложена. Инициализирующее выражение может быть сложным, важно, чтобы оно было вычислимым в момент его определения. Вот пример объявления констант:

```
/// <summary>
/// Константы
/// </summary>
public void Constants()
{
        const int SmallSize = 38, LargeSize =58;
        const int MidSize = (SmallSize + LargeSize)/2;
        const double pi = 3.141593;
        //LargeSize = 60; //Значение константы нельзя изменить.
        Console.WriteLine("MidSize= {0}; pi={1}",
                MidSize, pi);
}//Constants
```